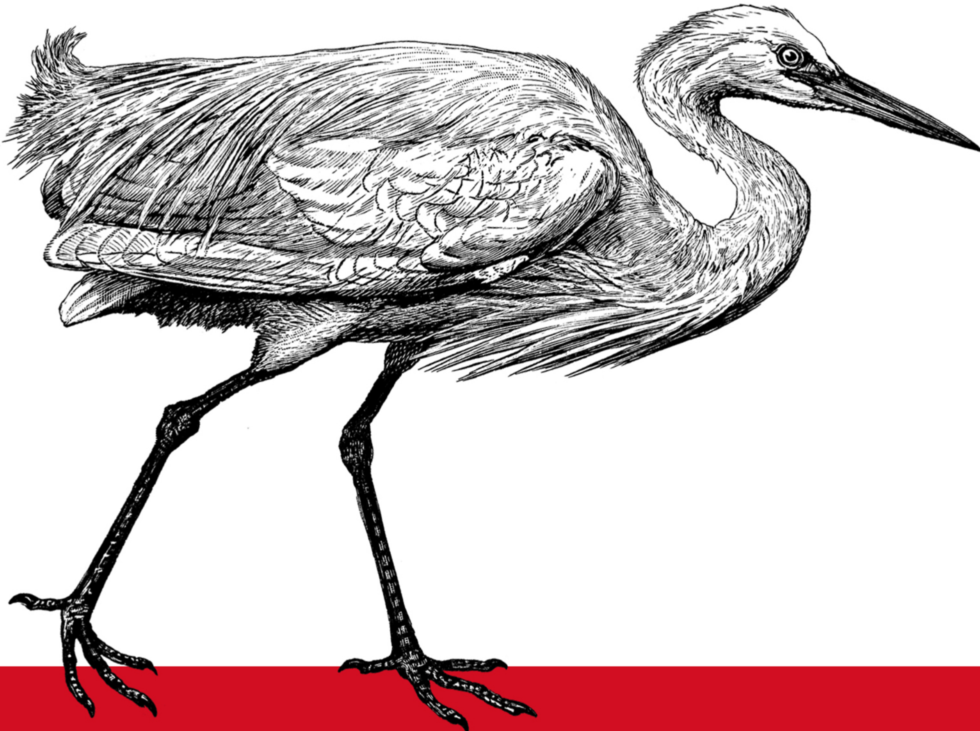
O'REILLY®

TURING 图灵程序设计丛书



Hadoop数据分析

Data Analytics with Hadoop

针对数据分析介绍分布式计算涉及的大量概念、
工具和技术，纵览Hadoop生态系统

[美] Benjamin Bengfort Jenny Kim 著
王纯超 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

王纯超

毕业于武汉工程大学软件工程专业，中南财经政法大学商务英语第二学位，拥有英语专业八级证书。工作5年，先后做过Java开发、PowerBuilder开发、ETL和VBA，因缘巧合进入大数据行业。热爱技术和英语，矢志为技术传播贡献绵薄之力。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵程序设计丛书

Hadoop数据分析

Data Analytics with Hadoop

[美] Benjamin Bengfort Jenny Kim 著
王纯超 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

Hadoop数据分析 / (美) 本杰明·班福特
(Benjamin Bengfort) , (美) 珍妮·基姆 (Jenny Kim)
著 ; 王纯超译. -- 北京 : 人民邮电出版社, 2018.4
（图灵程序设计丛书）
ISBN 978-7-115-47964-8

Ⅰ. ①H… Ⅱ. ①本… ②珍… ③王… Ⅲ. ①数据处
理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2018)第036682号

内 容 提 要

通过提供分布式数据存储和并行计算框架，Hadoop 已经从一个集群计算的抽象演化成了一个大数据的操作系统。本书旨在通过以可读且直观的方式提供集群计算和分析的概览，为数据科学家深入了解特定主题领域铺平道路，从数据科学家的视角介绍 Hadoop 集群计算和分析。本书分为两大部分，第一部分从非常高的层次介绍分布式计算，讨论如何在集群上运行计算；第二部分则重点关注数据科学家应该了解的工具和技术，意在为各种分析和大规模数据管理提供动力。

本书适合数据科学领域的从业人员，以及对数据分析感兴趣的研究人员。

◆ 著　[美] Benjamin Bengfort　Jenny Kim
译　王纯超
责任编辑　朱　巍
执行编辑　夏静文
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：14.25
字数：337千字　2018年 4 月第 1 版
印数：1 – 3 500册　2018年 4 月北京第 1 次印刷
著作权合同登记号　图字：01-2017-6475号

定价：69.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言 ix

第一部分　分布式计算入门

第 1 章　数据产品时代 2
- 1.1　什么是数据产品 2
- 1.2　使用 Hadoop 构建大规模数据产品 4
  - 1.2.1　利用大型数据集 4
  - 1.2.2　数据产品中的 Hadoop 5
- 1.3　数据科学流水线和 Hadoop 生态系统 6
- 1.4　小结 8

第 2 章　大数据操作系统 9
- 2.1　基本概念 10
- 2.2　Hadoop 架构 11
  - 2.2.1　Hadoop 集群 12
  - 2.2.2　HDFS 14
  - 2.2.3　YARN 15
- 2.3　使用分布式文件系统 16
  - 2.3.1　基本的文件系统操作 16
  - 2.3.2　HDFS 文件权限 18
  - 2.3.3　其他 HDFS 接口 19
- 2.4　使用分布式计算 20
  - 2.4.1　MapReduce：函数式编程模型 20

2.4.2 MapReduce：集群上的实现……22
2.4.3 不止一个 MapReduce：作业链……27
2.5 向 YARN 提交 MapReduce 作业……28
2.6 小结……30

第 3 章 Python 框架和 Hadoop Streaming……31

3.1 Hadoop Streaming……32
3.1.1 使用 Streaming 在 CSV 数据上运行计算……34
3.1.2 执行 Streaming 作业……38
3.2 Python 的 MapReduce 框架……39
3.2.1 短语计数……42
3.2.2 其他框架……45
3.3 MapReduce 进阶……46
3.3.1 combiner……46
3.3.2 partitioner……47
3.3.3 作业链……47
3.4 小结……50

第 4 章 Spark 内存计算……52

4.1 Spark 基础……53
4.1.1 Spark 栈……54
4.1.2 RDD……55
4.1.3 使用 RDD 编程……56
4.2 基于 PySpark 的交互性 Spark……59
4.3 编写 Spark 应用程序……61
4.4 小结……67

第 5 章 分布式分析和模式……69

5.1 键计算……70
5.1.1 复合键……71
5.1.2 键空间模式……74
5.1.3 pair 与 stripe……78
5.2 设计模式……80
5.2.1 概要……81
5.2.2 索引……85
5.2.3 过滤……90
5.3 迈向最后一英里分析……95
5.3.1 模型拟合……96
5.3.2 模型验证……97
5.4 小结……98

第二部分　大数据科学的工作流和工具

第 6 章　数据挖掘和数据仓储……102
6.1 Hive 结构化数据查询……103
6.1.1 Hive 命令行接口（CLI）……103
6.1.2 Hive 查询语言……104
6.1.3 Hive 数据分析……108
6.2 HBase……113
6.2.1 NoSQL 与列式数据库……114
6.2.2 HBase 实时分析……116
6.3 小结……122
第 7 章　数据采集……123
7.1 使用 Sqoop 导入关系数据……124
7.1.1 从 MySQL 导入 HDFS……124
7.1.2 从 MySQL 导入 Hive……126
7.1.3 从 MySQL 导入 HBase……128
7.2 使用 Flume 获取流式数据……130
7.2.1 Flume 数据流……130
7.2.2 使用 Flume 获取产品印象数据……133
7.3 小结……136
第 8 章　使用高级 API 进行分析……137
8.1 Pig……137
8.1.1 Pig Latin……138
8.1.2 数据类型……142
8.1.3 关系运算符……142
8.1.4 用户定义函数……143
8.1.5 Pig 小结……144
8.2 Spark 高级 API……144
8.2.1 Spark SQL……146
8.2.2 DataFrame……148
8.3 小结……153
第 9 章　机器学习……154
9.1 使用 Spark 进行可扩展的机器学习……154
9.1.1 协同过滤……156
9.1.2 分类……161
9.1.3 聚类……163
9.2 小结……166

第 10 章　总结：分布式数据科学实战……167
10.1　数据产品生命周期……168
10.1.1　数据湖泊……169
10.1.2　数据采集……171
10.1.3　计算数据存储……172
10.2　机器学习生命周期……173
10.3　小结……175
附录 A　创建 Hadoop 伪分布式开发环境……176
附录 B　安装 Hadoop 生态系统产品……184
术语表……193
关于作者……211
关于封面……211

前言

大数据已经成为一个流行词。人们用它来描述数据驱动型应用程序中的那些令人兴奋的新工具和新技术。这些应用程序正为我们带来崭新的计算方式。令统计学家懊恼的是，这一词语似乎被随意使用，其范围甚至包括在大型数据集上使用众所周知的统计技术进行预测。虽然大数据已经成为流行词，但事实上，现代分布式计算技术能分析的数据集远比过去那些“典型”方式能应对的数据集大得多，结果也更令人震撼。

然而，单纯的分布式计算并不等于数据科学。互联网带来了快速增长的数据集，这些数据集又能驱动预测模型（“更多的数据优于更好的算法”[1]），数据产品也因此成为了一种新型的经济范式。为大型跨域异构数据集建模所取得的巨大成功（例如 Nate Silver 通过大数据技术像使用魔法一般预测了 2008 年的美国大选结果），使很多人认识到了数据科学的价值，也为这个领域吸引了大量从业者。

通过提供分布式数据存储和并行计算框架，Hadoop 已经从集群计算的抽象演变成了大数据操作系统。Spark 正是基于这一理念构建的，它使数据科学家能更轻松地使用集群计算。然而，不了解分布式计算的数据科学家和分析人员可能会觉得这些工具是面向**程序员**的，而不是面向**分析人员**的。这是因为，我们需要从根本上转变管理数据和计算数据的思维方式，这样才能从串行模式转换到**并行**模式。

本书旨在通过**可读**且直观的方式介绍集群计算和分析，帮助数据科学家完成这一思维转换。我们将针对数据分析介绍分布式计算涉及的大量概念、工具和技术，为深入了解特定领域铺平道路。

本书目标

本书不会详细讲解 Hadoop（推荐 Tom White 的《Hadoop 权威指南》），也不是 Spark 入门资料（推荐 Holden Karau 等人所著的《Spark 快速大数据分析》[2]），当然更不是为了教你如

注 1：Anand Rajaraman,“More data usually beats better algorithms”(http://anand.typepad.com/datawocky/2008/03/more-data-usual.html), Datawocky, March 24, 2008.

注 2：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。——编者注

何进行分布式计算。本书将纵览 Hadoop 生态系统和分布式计算，旨在武装数据科学家、统计学家、程序员和对 Hadoop 感兴趣（但是对 Hadoop 的了解十分有限）的人。希望本书能成为你深入 Hadoop 世界的向导，助你找到最感兴趣的工具和技术。这可能是 Spark、Hive、机器学习、ETL（抽取、转换和加载）操作、关系数据库或者众多与集群计算相关的主题之一。

目标读者

人们经常把数据科学与大数据混为一谈。虽然为了达到良好的泛化效果，许多机器学习模型确实需要大型数据集，但即使是小型数据集也能支持模式识别。因此，数据科学软件类的图书大多关注易于在一台机器（尤其是内存容量多达几吉字节的机器）上分析的数据集。尽管大数据和数据科学非常适合协同工作，但是直到今天，与计算相关的图书还是将它们分开讨论。

本书以数据科学家为目标读者，旨在弥补这一隔阂。它将以数据科学的视角介绍 Hadoop 集群计算和分析。本书的关注点不是部署、运维或软件开发，而是常用分析、数据仓储技术和高阶数据流。

那么，什么人算是数据科学家呢？本书所说的数据科学家是指具有高超统计技能的软件开发人员，或者具有强大软件开发能力的统计学家。通常情况下，数据团队由三类数据科学家组成，分别是数据工程师、数据分析师和领域专家。

数据工程师指能构建或者使用高级计算系统的程序员或者计算机科学家。他们通常使用 Python、Java 或者 Scala 编程，熟悉 Linux、服务器、网络、数据库和应用程序部署。如果你是数据工程师，本书假设你能适应多进程编程、数据整理和数值计算。希望你在阅读本书后，能更了解如何在集群上部署应用程序，学会如何处理比单机在足够时间内能处理的数据集还要大得多的数据集。

数据分析师主要关注统计建模和探索性数据分析。在日常工作中，他们通常使用 R、Python 或者 Julia，熟悉数据挖掘和机器学习技术，比如回归、聚类和分类问题。数据分析师很可能通过采样处理过更大的数据集。我们将在本书中展示数据统计技术，处理比以往获取的数据量大得多的数据，从而构建预测能力既有广度又有深度的模型。

领域专家是团队里富有影响力、面向业务的成员。他们深入了解数据类型和所碰到的问题，理解数据带来的特定挑战，并寻求通过更好的方式利用数据应对新挑战。希望本书能够为他们提供一些业务决策思路，让当前的数据流更加灵活，并帮助他们理解怎样使用通用的计算框架来应对特定的领域挑战。

阅读方式

至今为止，Hadoop 已经有十多年的历史了，就技术而言，这已经是很长一段时间了。然而，摩尔定律仍然没有慢下来。10 年前，在数据中心使用廉价的机器集群远比为超级计算机编程简单。但现在，同样的廉价服务器要比以前强大约 32 倍，内存计算的开销也降低了很多。Hadoop 成为了大数据的操作系统，支持图形处理、类 SQL 查询和流处理等多种

计算框架。但这也给想要学习 Hadoop 的人带来了巨大的挑战——该从何学起?

本书篇幅简短的原因只有一个——想要尽可能简洁地覆盖多个方面。我们希望你通过两种方式阅读本书：一是快速通读全书，对 Hadoop 和分布式数据分析有大致了解；二是选择感兴趣的章节深入学习。本书以**易懂**为目的。我们通过简单的代码示例进行讲解，不一定需要你亲自实现和运行代码。本书是 Hadoop 和 Spark 领域的指导手册，对分析人员尤其如此。

内容概述

本书旨在带领你了解 Hadoop 生态系统，书中内容分为两部分：第一部分（第 1 章至第 5 章）宏观地介绍分布式计算，讨论**如何**在集群上运行计算；第二部分（第 6 章至第 10 章）侧重于介绍数据科学家应该具体了解的工具和技术，意在为各种分析和大规模数据管理提供**动力**。（第 5 章将从对分布式计算的讨论过渡到更加具体的工具和大数据科学流水线的实现。）每章的内容概述如下。

第 1 章 数据产品时代

介绍大数据和数据科学的结晶——数据产品，讨论创建数据产品背后的流程，说明数据分析的串行模型如何与分布式计算相契合。

第 2 章 大数据操作系统

概述 Hadoop 背后的核心概念，讲解为何集群计算既有益又复杂；主要着眼于 YARN 和 HDFS，详细讨论 Hadoop 体系架构，讲解与分布式存储系统的交互，为分析大型数据集作准备。

第 3 章 Python 框架和 Hadoop Streaming

介绍分布式计算的基本编程抽象 MapReduce。然而，MapReduce 的 API 是用 Java 编写的，这不是一种在数据科学家间流行的编程语言。因此，这一章专注于介绍如何通过 Hadoop Streaming 使用 Python 编写 MapReduce 作业。

第 4 章 Spark 内存计算

虽然理解 MapReduce 对理解分布式计算和编写高性能的批处理作业（如 ETL）十分重要，但是 Hadoop 集群上的日常交互和分析却通常都是使用 Spark 完成的。这一章将介绍 Spark，以及如何使用 Python 编写 Spark 应用程序，并通过 PySpark 以交互方式在 YARN 上运行，或者在集群模式下运行。

第 5 章 分布式分析和模式

通过展示设计模式和并行分析算法，从实践的角度研究怎样编写分布式数据分析作业。开始阅读这一章之前，你应该已经了解编写 Spark 和 MapReduce 作业的原理。读完这一章，你应该能轻松实现它们。

第 6 章 数据挖掘和数据仓储

介绍分布式环境下的数据管理、数据挖掘和数据仓储，特别是与传统数据库系统密切相关的方面。这一章重点介绍 Hive 和 HBase，它们分别是 Hadoop 最流行的基于 SQL 的查询引擎和 NoSQL 数据库。数据整理是数据科学流水线的第二步，但是数据需要被采集到某处。这一章还将探索怎样管理大型数据集。

第 7 章 数据采集

考虑到数据的容量和速度，如何将数据导入分布式系统并用于计算可能才是最大的挑战之一。这一章将研究从关系数据库获取数据的批量加载工具 Sqoop 以及更灵活的 Apache Flume，后者用于获取日志和来自网络的其他非结构化数据。

第 8 章 使用高级 API 进行分析

研究用于编写复杂 Hadoop 和 Spark 应用程序的高阶工具，尤其是 Apache Pig 和 Spark 的 DataFrame API。第一部分将讨论 MapReduce 和 Spark 分布式作业的实现过程，以及怎样从数据流的角度看待算法和数据流水线。Pig 让你无须使用 MapReduce 实现底层细节，从而能更轻松地描述数据流。Spark 提供了多个集成模块，能无缝结合过程式处理与关系查询，为强大的分析定制打开了大门。

第 9 章 机器学习

大数据的多数益处都是在机器学习中得以实现的——更加广泛的特征和输入空间让模式识别技术更加有效和个性化。这一章将介绍分类、聚类和协同过滤，但并不会详细讨论建模，而是使用 Spark 的 MLlib 让你上手可扩展机器学习技术。

第 10 章 总结：分布式数据科学实战

完整呈现分布式数据科学，把前面章节中单独讨论的工具与技术结合起来。数据科学不是单一的活动，而是一个生命周期，涉及数据的采集、整理、建模、计算和操作化。这一章将从整体上讨论分布式数据科学的架构和工作流。

附录 A 创建 Hadoop 伪分布式开发环境

附录 A 将指导你在本机上搭建一个开发环境，从而编写分布式作业。如果你没有集群可用，附录 A 是运行本书示例至关重要的准备工作。

附录 B 安装 Hadoop 生态系统产品

附录 B 是附录 A 的延伸，将提供本书讨论的众多生态系统工具和产品的安装指导。尽管附录 A 提供了安装服务的常用方法，但附录 B 专门为安装服务（用来运行书中示例，你在阅读的过程中会遇到它们）的过程中会遇到的问题提供指导。

你看，这么薄的一本书却涵盖了这么多主题。希望以上这些内容足以吸引你继续阅读下去。

编程和示例代码

随着 Hadoop 的分布式计算变得更加成熟和集成化，并行计算正在向更丰富的分析体验转变。例如，大数据生态系统的最新成员 Spark 提供了 4 种语言的编程 API，更方便那些习惯于使用数据框、交互式 notebook 和解释型语言的数据科学家使用。Hive 和 SparkSQL 以 SQL 语法形式提供了另外一种为人们所熟知的领域专用语言（domain-specific language，DSL），专门针对分布式集群上的数据查询。

因为本书的目标读者是数据科学家，所以我们大多选择使用 Python 来实现示例。Python 是通用的编程语言，拥有丰富的分析包（例如 Pandas 和 Scikit-Learn），在数据科学领域占有一席之地。不幸的是，Hadoop 的主要 API 通常都是以 Java 编写的，那些 Python 示例让我

们大费周章，但是大多数时候，我们会用更实际的方式来阐明思想。因此，本书中的代码要么是使用 Python 和 Hadoop Streaming 的 MapReduce，要么是使用 PySpark API 的 Spark 代码，或者是讨论 Hive、Spark SQL 时的 SQL 代码。希望这能让更多读者感觉简明易懂。

GitHub仓库

你可以在我们的 GitHub 仓库（https://github.com/bbengfort/hadoop-fundamentals）找到本书完整且可执行的示例代码。这个仓库也包含了我们的 Hadoop 视频教程“Hadoop Fundamentals for Data Scientists”[3] 的代码。

为了在纸质版中呈现代码并更清楚地解释过程，我们走了捷径，省略了代码的细节。例如，通常都省略了 `import` 语句——这意味着简单的复制粘贴无法奏效。然而，你可以使用仓库中的例子，它们是完整且可执行的代码，并附有相应的注释。

但要注意，仓库是持续更新的，可以查阅 README 文件以了解更新情况。你当然可以 fork 仓库，更改代码以在你自己的环境中运行——我们强烈推荐你这样做！

执行分布式作业

Hadoop 开发人员通常在“伪分布式模式”下使用“单节点集群”进行开发任务。该集群通常是一个运行着虚拟服务器环境的虚拟机，环境中运行着多个 Hadoop 守护进程。你可以在主开发工具里使用 SSH 访问该虚拟机，就像访问 Hadoop 集群一样。为了创建虚拟环境，你需要某种虚拟化软件，例如 VirtualBox（https://www.virtualbox.org）、VMWare（http://www.vmware.com/products/desktop-virtualization）或者 Parallels（http://www.parallels.com）。

附录 A 讨论怎样设置以伪分布式模式运行 Hadoop、Hive 和 Spark 的 Ubuntu x64 虚拟机。你也可以使用一些 Hadoop 发行版（例如 Cloudera 和 Hortonworks）提供的预先配置好的虚拟环境。如果你有想用的虚拟机环境，那么我们建议你下载它。如果你想了解更多的 Hadoop 操作，就请自己配置吧！

还有一点，因为 Hadoop 集群是在开源软件上运行的，所以你需要了解 Linux 和命令行。本书讨论的虚拟机通常都是通过命令行访问的，书中的许多例子都描述了通过命令行与 Hadoop、Spark、Hive 和其他工具交互的过程。命令行是分析人员不愿使用这些工具的一个主要原因。然而，学习命令行对你大有帮助，它也并不可怕。我们建议你学习一下！

使用示例代码

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

注 3：http://shop.oreilly.com/product/0636920035183.do.

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN，比如“*Data Analytics with Hadoop* by Benjamin Bengfort and Jenny Kim (O'Reilly). Copyright 2016 Benjamin Bengfort and Jenny Kim, 978-1491-91370-3”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

反馈及作者联系方式

关于本书的评论和技术性问题，请发送电子邮件至 bookquestions@oreilly.com。

工具和技术的变化非常快，在大数据领域尤其如此。不幸的是，很难保证一本书（特别是纸质版）时刻跟上潮流。我们希望本书能够在未来继续为你服务，但如果你发现有变更让书中的示例无法运行或导致代码问题，请联系我们。

如果有关于代码或示例的问题，请在 GitHub 上（https://github.com/bbengfort/hadoop-fundamentals/issues/）提交问题，这是与我们联系的最佳方式。你也可以发送电子邮件到 hadoopfundamentals@gmail.com，我们会尽快回复。非常感谢你提供积极且具有建设性的反馈！

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几百家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。[4] 本书的网页是：

http://shop.oreilly.com/product/0636920035275.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

我们要感谢本书的审校者，你们在漫长的写作过程中持续提出颇具建设性的反馈和批评。感谢 Marck Vaisman，你从向数据科学家讲授 Hadoop 的角度阅读本书；特别感谢 Konstantinos Xirogiannopoulos 在忙碌的研究之余，志愿向我们提供清晰、有益且积极的评论，我们很高兴收到这样的评论。

我们还要感谢 O’Reilly 每位富有耐心、坚持不懈的编辑。在本书写作之初，我们走了一些弯路，是 Meghan Blanchette 带领我们一路走了过来，她一直支持着我们。很遗憾，在本书还未截稿时，她已离开 O’Reilly 去追求更好的事业。当 Nicole Tache 接替她并成功将我们带回正轨时，我们特别高兴。Nicole 引导我们完成写作，没有她，就没有本书。她有一项特殊的本领，总能在关键时刻发来让人看了就舒心的邮件，让工作如期完成。与 O’Reilly 每个人的合作都很愉快，特别是 Marie Beaugureau、Amy Jollymore、Ben Lorica 和 Mike Loukides，感谢你们给予的建议和鼓励。

在华盛顿，District Data Labs 的工作人员给予了我们巨大的支持。我们要特别提到 Tony Ojeda、Rebecca Bilbro、Allen Leis 和 Selma Gomez Orr，你们以各种方式支持本书，包括第一个购买早期版本、提供反馈、审查代码并询问完成时间，这都鼓励我们专心写作！

如果没有 Hadoop 社区中各位了不起的贡献，本书就不可能诞生，Jenny 更是有幸每天在 Cloudera 与这些志同道合者一起工作。特别感谢 Hue 团队，他们为提供最好的 Hadoop 用

注 4：也可以通过图灵社区获取相关信息：http://www.ituring.com.cn/book/1944。——编者注

户体验所做出的贡献和表现出的热情既超乎寻常又鼓舞人心。

感谢我们的家人，特别是 Benjamin 的父母 Randy Bengfort 和 Lily Bengfort，以及 Jenny 的父母 Wung Kim 和 Namoak Kim，感谢你们一直以来的鼓励、爱与支持。我们的父母向我们灌输了相互学习和探索的热情，因此我们钻研了许多方面。他们使我们拥有坚韧的精神和恒心，总能让我们找到实现目标的方法。

最后，感谢我们的伴侣 Patrick 和 Jacquelyn，感谢你们一直支持我们。我们中的谁可能说过："再写一本书，我的婚姻就要结束了。"诚然，在写作的最后阶段，他们都不愿意听到我们仍然在努力。但如果没有他们，我们的书就无法写成（"婚姻不在，书也就不在了"）。在我们通过视频电话商定细节和重写时，Patrick 和 Jacquelyn 总是和颜悦色。他们甚至阅读了部分内容，提供建议，在各方面都提供了帮助。在这之前，我们都没有写过书，也不知道会面临什么问题。现在我们知道了，也很高兴他们一直在身边支持我们。

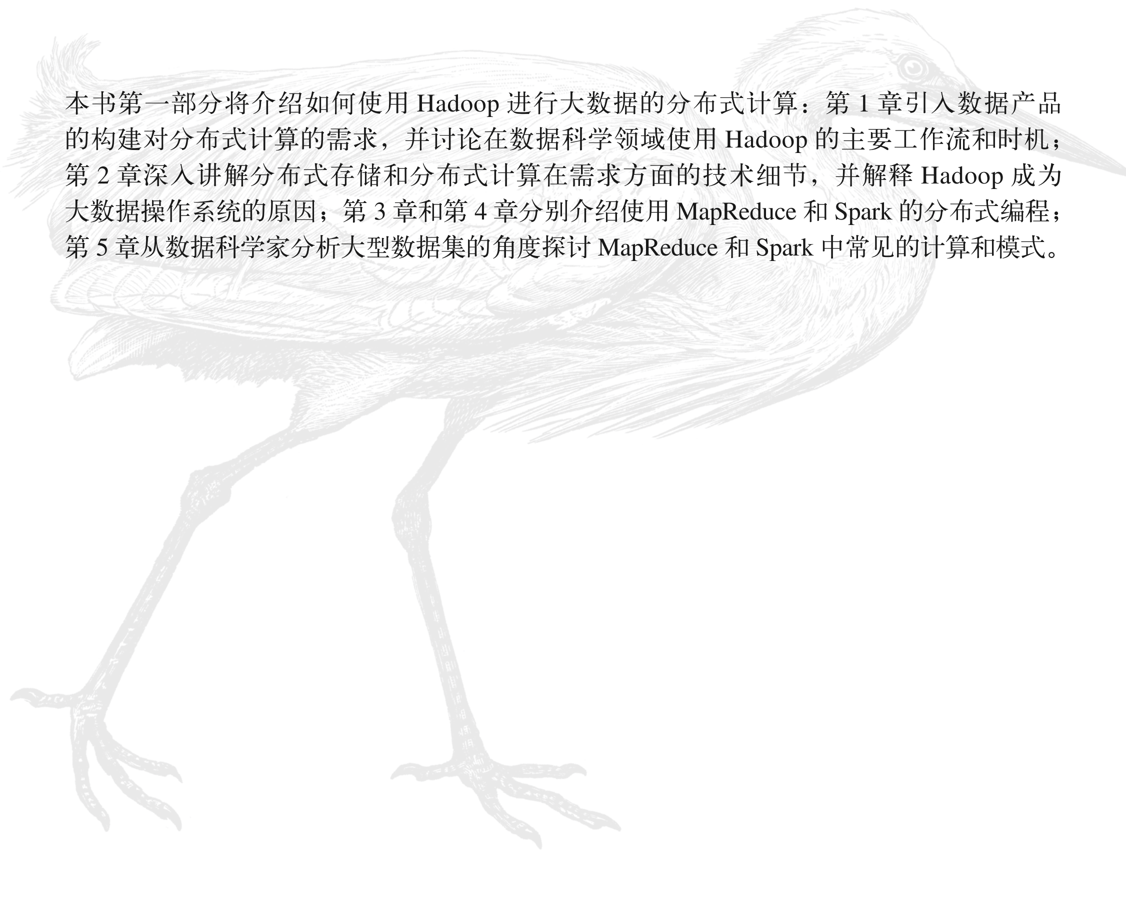
电子书

扫描如下二维码，即可购买本书电子版。

第一部分

分布式计算入门

本书第一部分将介绍如何使用 Hadoop 进行大数据的分布式计算：第 1 章引入数据产品的构建对分布式计算的需求，并讨论在数据科学领域使用 Hadoop 的主要工作流和时机；第 2 章深入讲解分布式存储和分布式计算在需求方面的技术细节，并解释 Hadoop 成为大数据操作系统的原因；第 3 章和第 4 章分别介绍使用 MapReduce 和 Spark 的分布式编程；第 5 章从数据科学家分析大型数据集的角度探讨 MapReduce 和 Spark 中常见的计算和模式。



第 1 章

数据产品时代

我们生活在一场信息革命之中。和任何经济革命一样，它对社会、学术界和商界都造成了极具变革性的影响。眼前这场革命由网络通信系统和互联网驱动，其独特之处在于创造了大量有价值的新材料——数据，并将所有人转变成了消费者和生产者。每天都有海量的数据生成，数据也日益影响着生活的各个方面，从吃的食物，到社交活动，再到工作和娱乐的方式。同样，我们也对产品和服务有了合理的期望，比如高度个性化，或能针对我们的身体、生活和职业进行优化。这为一项崭新的信息技术创造了市场——**数据产品**。

过剩的数据集与机器学习算法快速敏捷地结合，这不仅改变了人们与日常事物的交互方式，也改变了彼此打交道的方式，因为这一结合经常带来立竿见影且富有新意的成果。的确，大量的模型和数据源似乎带来了无穷无尽的创新，围绕“大数据”这一热词的趋势正与此有关。

数据产品通过数据科学工作流创建；具体来说，是将模型（通常是预测性的或推断性的）应用于特定领域的数据集。虽然创新的潜力是巨大的，但是发现数据源并正确建模或挖掘模式需要科学性或实验性的思维模式，而程序员和分析人员通常不具备这一能力。也正是出于这个原因，雇用博士确实省时省力——他们经过必要的分析和实验训练，结合编程，基本立即就能成为数据科学专业人才。当然，我们不可能都是博士。因此，本书提出了一个将 Hadoop 用于大规模数据科学的教学模型，并且将其作为构建应用程序的基础，这些应用程序正是（或可以成为）数据产品。

1.1　什么是数据产品

这个问题的传统答案通常是：“任何将数据和算法结合起来的应用程序。”[1] 但坦白说，如果

注 1：Hillary Mason and Chris Wiggins,“A Taxonomy of Data Science”(http://www.dataists.com/2010/09/a-taxonomy-of-data-science/), Dataists, September 25, 2010.

你编写的软件没有将数据与算法结合在一起，那么你在做什么呢？毕竟数据可是编程界的"货币"！具体来说，数据产品就是数据与用于推断或预测的统计算法的结合。许多数据科学家也是统计学家，统计方法论是数据科学的核心。

根据这个定义，Amazon 的推荐系统就是一个数据产品。Amazon 会检查你购买的商品，并根据其他用户类似的购买行为作出推荐。在这种情况下，将订单历史数据与推荐算法相结合，就能预测你将来可能购买什么。Facebook 的"你可能认识的人"也是一个数据产品，因为它"基于共同的好友、工作、教育信息以及许多其他因素向你推荐好友"——这本质上是结合社交网络数据与图算法来推断社区成员。

这两个产品分别在零售业和社交网络领域具有革命性意义，但它们看上去与其他 Web 应用程序没什么不同。的确，简单地将数据产品定义为数据与统计算法的结合，似乎将其限制为了单一的软件类型（如 Web 应用程序），这很难成为一股革命性的经济力量。虽然可以说 Google 或其他公司是庞大的经济力量，但是仅将收集庞大 HTML 语料库的 Web 爬虫与 PageRank 算法结合却不能创造数据经济。众所周知，搜索在经济活动中起到了重要作用，因此上述定义一定有缺失。

Mike Loukides 认为，数据产品不仅仅是"数据驱动的应用程序"。虽然博客、电子商务平台以及大多数 Web 应用程序和移动应用程序都依赖于数据库和数据服务（如 RESTful API），但它们只使用数据，其本身并不构成数据产品。他对数据产品的定义如下。[2]

> 数据应用程序从数据本身获取价值，然后创造更多数据。它不仅仅是带有数据的应用程序，而是数据产品。

这是一场革命，数据产品则是经济引擎。它从数据中获取价值，作为回报，再产生更多的数据和价值。它产生的数据可为自己添加燃料（我们终于实现了永动机），或者催生其他数据产品，这些数据产品从生成的数据中获取价值。正是这一过程带来了信息过剩和信息革命。更重要的是，这种**生成**效应让我们通过数据过上了更好的生活，因为更多的数据产品意味着更多的数据，更多的数据意味着更多的数据产品，循环往复。

有了这个更具体的定义，我们就可以更进一步，将数据产品描述为一个**从数据中学习**、**自适应**并且**广泛适用**的系统。根据这个定义，Nest 恒温器算是数据产品：它从传感器数据中获取价值，决定制热或制冷，并且收集新的传感器数据以检验调节效果。由斯坦福大学的无人驾驶团队研制的无人驾驶汽车也属于这一类。该团队通过算法实现机器视觉并模拟驾驶员行为，因此车辆在行驶时能产生更多导航数据和传感器数据，这些数据可用于改进驾驶平台。此外，由 Fitbit、Withings 和许多其他公司发起的"量化自我"产品意味着数据可以影响人类行为；智能电网意味着数据会影响你的效能。

数据产品是自适应且广泛适用的经济引擎。它从数据中获取价值，并通过影响人类行为或通过基于新数据的推断或预测产生更多数据。数据产品不只是 Web 应用程序，它正迅速成为现代社会几乎每一个经济活动领域的重要组成部分。数据产品能够发现人类活动中的个体模式，所以它能推动决策，由它引发的行动和影响也会被记录为新的数据。

注 2：Mike Loukides, "What is Data Science?" (https://www.oreilly.com/ideas/what-is-data-science), O'Reilly Radar, June 2, 2010.

1.2　使用Hadoop构建大规模数据产品

Josh Wills 经常被引用的推文[3]给我们提供了以下定义。

> 数据科学家（名词）：指比所有软件工程师更擅长统计学，并且比所有统计学家更擅长软件工程的人。

当然，这与数据产品仅仅是数据与统计算法的结合这一想法十分吻合。软件工程和统计学知识都是数据科学的基础。然而，在一个需要产品从数据中获取价值并产生新数据的经济体系中，构建数据产品其实就是数据科学家的工作。

Harlan Harris 提供了有关数据产品的更多细节[4]：它们建立在数据、领域知识、软件工程和分析技术的交叉点上。由于数据产品是系统，因此构建它们需要工程技能，通常是软件工程方面的技能；由于它们由数据驱动，因此拥有数据是必要条件；领域知识和分析技术是用于构建数据引擎的工具，通常通过实验完成，因此是数据科学的"科学"部分。

由于需要使用实验方法学，因此大多数数据科学家会采用典型的分析工作流：采集 → 整理 → 建模 → 报告和可视化。然而，这种所谓的**数据科学流水线**完全由人力驱动，再辅以脚本语言（如 R 和 Python）的使用。流水线的每一个环节都需要人类的知识和分析技能，意在产生独特且不可泛化的结果。虽然这个流水线是很好的统计和分析基础框架，但它不能满足构建数据产品的需求，特别是当想从中获取价值的目标数据大到无法在一台笔记本电脑上处理时。随着数据越来越多、越来越多变、产生的速度越来越快，自动获取有用信息而无须人工干预的工具也变得越来越重要。

1.2.1　利用大型数据集

直觉告诉我们，观测越多，数据就越多——这真让人喜忧参半。人类拥有发现大规模模式的卓越能力（我们以森林和林中空地作为隐喻）。理解数据的认知过程涉及概览数据，深入研究具体层面的细节，然后再回到概览角度。这个过程中的细节并不一定可靠，因为细粒度（隐喻中的叶子、分枝或单棵树木）会限制我们的理解能力。多数数据既可能是模式和信号，也可能是噪声和干扰。

通过聚合和索引描述数据，或者直接对数据建模，统计方法使我们能够处理掺杂着噪声和信号的数据。虽然这些技术能帮助我们理解数据，但是它们以牺牲计算粒度为代价，例如有意义的罕见事件可能会被模型排除。兼顾罕见事件的统计技术能利用计算机同时跟踪多个数据点，但也需要更多的计算资源。因此，传统的统计方法会对较大的数据集采取抽样方法，用较小的数据子集替代总体。样本越大，模型包括罕见事件且能将其捕获的可能性就越大。

随着收集数据的能力越来越高，我们对通用性也有了更大的需求。过去十年间，由于数据和机器学习算法的紧密结合，新颖的成果纷纷问世，数据科学得到了空前的发展。智能电网、"量化自我"、移动技术、传感器和互联家庭要求我们应用个性化的统计推断。规模不

注 3：https://twitter.com/josh_wills/status/198093512149958656.

注 4：Harlan Harris, "What Is a Data Product?", Analytics 2014 Blog, March 31, 2014.

仅与数据量有关，也与需要探索多少方面有关——就好像森林中的每棵树一样。

Google 的两篇论文描述了一个完整的分布式计算系统；Hadoop 是其开源实现，它将我们带入了大数据时代。然而，分布式计算和分布式数据库系统并不是新的话题。在那两篇论文发表之前，与 Hadoop 的计算能力相当的数据仓库系统就早已存在于工业界和学术界。Hadoop 之所以与众不同，一方面是因为数据处理能带来经济效益，另一方面是因为 Hadoop 是一个平台。但是真正使 Hadoop 独树一帜的原因其实是它出现的时机——恰恰在一个需要大规模数据分析解决方案的时刻，它问世了。而且它不仅能分析总体的统计数据，还能获得个体级别的通用性和洞察力。

1.2.2　数据产品中的Hadoop

一开始，Hadoop 的使用者是那些面临大数据挑战的大公司，比如 Google、Facebook 和 Yahoo。然而，Hadoop 之所以这么重要，以及促使你拿起本书的原因，恰恰是因为面临数据挑战的不再只是科技巨擘。大大小小的商业机构和政府机构——从企业到创业公司，再到联邦机构和市政府，甚至是每个人，都面临着数据挑战。计算资源变得廉价且唾手可得——就像 PC 时代的黑客在车库里使用手边的电子产品搞创新，现在的创业公司使用 10 节点～ 20 节点的小集群在数据探索上搞创新。云计算资源（如 Amazon EC2 和 Google Compute Engine）使数据科学家可以及时、按需地访问大规模集群，而且成本较低，也无须进行数据中心管理。Hadoop 使大数据计算更贴近大众，也更平易近人，下面的例子说明了这一点。

2011 年，Lady Gaga 发行了她的专辑 *Born This Way*，这个事件为社交媒体带来了约 1.3 万亿条信息，包括点赞、推文、图像和视频。Lady Gaga 的经纪人 Troy Carter 马上发现了一个将粉丝聚集起来的机会。经过大量的数据挖掘工作，他成功将 Twitter 和 Facebook 上的数百万粉丝聚集到了 LittleMonsters.com 这个只针对 Lady Gaga 的小社交网络中。该网站的成功促使 Backplane（现在叫 Place）诞生，这是一个用于生成和管理由小型社区驱动的社交网络的工具。

2015 年，纽约市警察局安装了一个价值 150 万美元的声学传感器网络，名叫 ShotSpotter。该系统能够检测与爆炸或枪击相关的脉冲声，使应急响应人员能够快速响应在布朗克斯区发生的事件。重要的是，这个系统还很智能，可以预测是否会发生后续的枪击事件及其大致位置。ShotSpotter 系统发现，自 2009 年以来，有超过 75%的枪击事件没有报告给警察。

“量化自我”运动越来越受欢迎，各家公司也一直努力在消费者中广泛普及可穿戴技术设备、个人数据收集设备，甚至是基因测序仪器。2012 年，美国的《平价医疗法案》规定健康计划对电子病历实施标准化、安全、保密的共享方法。互联家庭、移动设备以及其他个人传感器每天都在产生大量个人数据，这引发了人们对隐私的关注。2015 年，英国研究人员创建了 *Hub of All Things*（HAT）。这是一项个性化的数据集合技术，用于处理“谁拥有你的数据”这一问题，并为个人数据的聚合提供技术解决方案。

传统上，大规模的个人数据分析一直属于社交网络的范畴，如 Facebook 和 Twitter。但幸好有了 Place，大型社交网络现在成为了个人品牌和艺术家的诞生之地。每个城市面临的数据挑战都不一样，尽管针对典型城市的泛化可以满足许多分析的需求，但是新的数据挑

战仍然不断出现，对每个城市分别进行研究势在必行。（比如工业、航运或天气对声学传感器网络有什么影响？）怎样使技术为消费者提供价值，在使用他们的个人医疗记录时不侵犯他们的隐私，避免与其他人的记录聚合？怎样使个人医疗诊断数据挖掘变得更安全？

数据产品的出现正是为了切实回答这些问题。Place、ShotSpotter、“量化自我”产品和 HAT 等通过提供应用程序平台和决策资源供人们采取行动，从数据中获取价值并产生新数据。它们提供的价值是明确的，但要处理数万亿的点赞数据和数百万个麦克风生成的大量数据集，或者我们每天生成的海量个人数据，传统的软件开发工作流无法应对这一挑战。大数据工作流和 Hadoop 使这些应用程序成为可能并且可个性化。

1.3 数据科学流水线和Hadoop生态系统

数据科学流水线是一种教学模型，用于教授对数据进行全面统计分析所需的工作流，如图 1-1 所示。在每个环节中，分析人员要转换初始数据集，然后从各种数据源增强或采集数据，再通过描述性或推断性的统计方法将数据整理为可以计算的正常形式，最后通过可视化或报告的形式生成结果。这些分析过程通常用于回答特定问题，或用于调查数据与某些业务实践间的关系，以进行验证或决策。

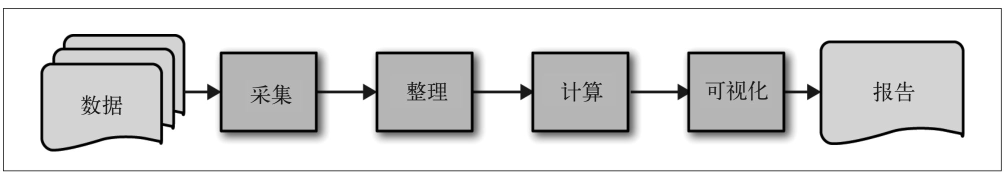


图 1-1：数据科学流水线

这个原始的工作流模型引领了大多数早期的数据科学思想。最初关于数据科学应用程序的讨论围绕着如何创建有意义的信息可视化——这也许令人意外，主要是因为这个工作流旨在生成帮助人们进行决策的依据。通过对大型数据集的聚合、描述和建模，人们能够更好地根据模式（而不是单个数据点）作出判断。数据可视化是新生的数据产品，它们从数据中产生价值，帮助人们基于学习到的内容采取行动，然后再从这些行动中生成新数据。

然而，面对呈指数增长的数据量和数据增长速度，这种以人力驱动的模型并不是一个可扩展的解决方案，这也正是许多企业都为之抓狂的原因。根据预测，到 2020 年，我们每年生成和复制的数据将达到 44ZB，即 44 万亿 GB。[5] 即使实际规模只达到预测规模的一小部分，手动的数据准备和挖掘方法也根本无法及时提供有意义的信息。

除了规模上的局限，这种以人为中心的单向工作流也不能有效地设计能够学习的自适应系统。机器学习算法已经广泛应用于学术界之外，非常符合数据产品的定义。因为模型会拟合现有的数据集，所以这些类型的算法可以从数据中获取价值，然后通过对新的观察值作出预测来产生新的数据。

注 5：EMC Digital Universe with Research & Analysis by IDC, “The Digital Universe of Opportunities” (https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm), April 2014.

如果要创建一个框架，支持构建可扩展和可自动化的解决方案，从而能解释数据和生成有用的信息，就必须修改数据科学流水线，使其包含机器学习方法的反馈循环。

大数据工作流

考虑到可扩展性和自动化的目标，我们可以将人力驱动的数据科学流水线重构为包括**采集**、**分段**、**计算**和**工作流管理**这 4 个主要阶段的迭代模型（如图 1-2 所示）。与数据科学流水线一样，这种模型其实就是采集原始数据并将其转换为有用的信息。关键的区别在于，数据产品流水线是在操作化和自动化工作流的步骤中构建起来的。通过将采集、分段和计算这 3 个步骤转换为自动化工作流，最终产生可重用的数据产品。工作流管理步骤还引入了反馈流机制，来自其中一个作业执行的输出可以自动作为下一次迭代的数据输入，因此为机器学习应用程序提供了必要的自适应框架。

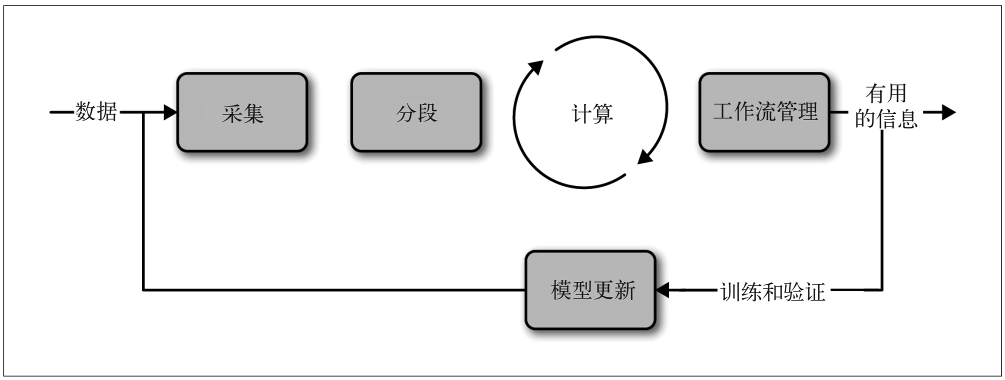


图 1-2：大数据流水线

采集阶段既是模型的初始化阶段，也是用户和模型之间的应用交互阶段。在初始化期间，用户指定数据源的位置或标注数据（另一种数据采集形式）；在交互期间，用户消费模型的预测结果并提供用于巩固模型的反馈。

分段阶段是转换数据的阶段，使其变为可消费的形式并存储起来，从而能够用于处理。本阶段还负责数据的归一化和标准化，以及一些计算数据存储中的数据管理工作。

计算阶段是真正“干活”的阶段，主要负责挖掘数据以获取有用的信息，执行聚合或报告，构建用于推荐、聚类或分类的机器学习模型。

工作流管理阶段执行抽象、编排和自动化任务，使工作流的各步骤可用于生产环境。此步骤应能产生自动按需运行的应用程序、作业或脚本。

Hadoop 已经演变成了包含各种工具的生态系统，可以实现上述流水线的部分环节。例如，Sqoop 和 Kafka 可用于数据采集，支持将关系数据库导入 Hadoop 或分布式消息队列，以进行按需处理。在 Hadoop 中，像 Hive 和 HBase 之类的数据仓库提供了大规模的数据管理机会；Spark 的 GraphX、MLlib 或 Mahout 库提供了分析包，供大规模计算和验证使用。在本书中，我们将探索 Hadoop 生态系统的许多组件，并了解它们如何融入整个大数据流水线。

1.4 小结

在过去十年间，关于“什么是数据科学”的讨论发生了巨大变化——从纯分析到与可视化相关的方法，再到如今数据产品的创建。数据产品是**使用数据训练、自适应且广泛适用**的经济引擎，从数据中获取价值并产生新的数据。数据产品引发了一次信息经济革命，改变了小企业、技术创业公司、大型组织甚至政府机构看待其数据的方式。

本章描述了数据科学流水线原始教学模型的一个改良版本，并提出了数据产品流水线。数据产品流水线是迭代的，包括两个阶段：构建阶段和运行阶段（包括 4 个阶段：交互、数据、存储和计算）。这种架构可以有条不紊地执行大规模的数据分析，保留了实验、人与数据产品间的交互。而且当围绕数据产品构建的应用程序很大时，它还能支持部分环节的自动化。希望这个流水线可以帮你了解数据产品生命周期的大体框架，也能成为探索更多创新项目的基石。

因为本书是从数据科学家的角度探讨分布式计算和 Hadoop，所以我们认为，Hadoop 的作用是从大量不同来源采集多种形式的数据（其中包含大量实例、事件和类），并将其转换为有价值的事物——数据产品。

第2章 大数据操作系统

数据团队通常是由 5 至 7 名成员组成的小型团队，他们通过敏捷方法实现**由假设驱动**的工作流。虽然数据科学家通常认为自己是掌握了数据方面大量技能的通才[1]，但是他们往往只精于软件、统计或领域专业知识其中之一。因此，数据团队的成员分为三大类：**数据工程师**负责数据的传输和原理的实践，其工作通常涉及软件和计算资源；**数据分析师**专注于探索和解释数据，并创建用于推断或预测的数据产品；**领域专家**提供过程和应用程序方面的专业知识以解决问题。

考虑到分布式计算的技术本质，使用 Hadoop 的数据团队常常将数据科学的重中之重放在数据工程方面。相较于基于实例的方法，大数据集更适合采用基于聚合的方法，用于分布式机器学习和统计分析的大型工具集也已经存在。因此，大多数关于 Hadoop 的资料针对的是软件开发人员，他们通常精通 Java——编写 Hadoop API 所用的软件语言。此外，培训材料也倾向于关注 Hadoop 的架构方面，因为这些方面展示了 Hadoop 之所以能成功处理大型机器学习等任务的创新本源。

本书的重点是如何使用 Hadoop 进行分析，而不是如何操作 Hadoop。然而，只有对分布式计算和存储的工作原理有所了解，才能更全面地了解如何使用 Hadoop 构建用于数据处理的算法和工作流。本章将展示 Hadoop 作为**大数据操作系统**的一面，通过两个主要组件——分布式文件系统 HDFS（Hadoop Distributed File System）以及负载和资源管理器 YARN（Yet Another Resource Negotiator）——概述 Hadoop 的原理。本章还将演示如何使用命令行与 HDFS 交互，并执行一个 MapReduce 作业。读完本章后，你应该能够轻松地与集群进行交互，并能执行本书其余部分的示例。

注 1：Harris, Harlan, Sean Murphy, Marck Vaisman, *Analyzing the Analyzers* (http://oreil.ly/1PxPrNg)(O’Reilly, 2013).

2.1 基本概念

为了执行大规模计算，Hadoop 将涉及大型数据集的分析计算分发给许多机器，每台机器同时对各自的数据块进行运算。分布式计算不是新的概念，但它是一项技术挑战，需要开发分布式算法，管理集群中的机器，并实现网络和架构细节。具体来说，分布式系统必须满足以下要求。

容错性
: 如果一个组件失败，不应导致整个系统出现故障，系统应能降级到较低性能状态。如果失败的组件恢复了，它应该能够重新加入系统。

可恢复性
: 发生故障时，不应丢失数据。

一致性
: 一个作业或任务的失败不应该影响最终的结果。

可扩展性
: 负载的增加（更多的数据或更多的计算）导致性能下降，而不是出现故障；资源的增加应使容量按比例增加。

Hadoop 通过以下几个抽象概念来满足上述要求。（在正确实现的情况下，这些概念定义了集群如何管理数据存储和分布式计算；此外，了解这些概念成为 Hadoop 架构基本前提的原因有助于理解其他概念，如数据流水线和分析数据流。）

- 数据添加到集群后即刻被分发出去，并存储在多个节点上。最好用节点处理本地存储的数据，以将网络流量最小化。
- 数据存储在固定大小（通常为 128MB）的块中，每个块跨系统多次复制，以提供冗余和数据安全。
- 通常将计算作为一个作业。作业被分解成任务，每个节点针对单个数据块执行任务。
- 编写作业时，不需要考虑网络编程、时间或底层的基础设施，从而允许开发人员专注于数据和计算，而不是分布式编程细节。
- 系统应该以透明的方式将节点之间的网络流量最小化。每个任务应该是独立的，并且节点也不应在处理期间彼此通信，以确保没有会导致死锁的进程间依赖。
- 作业通常通过任务冗余来容错，这使得单个节点或任务的失败不会导致最终的计算结果不正确或不完整。
- 主程序将工作分配给 worker 节点，这使得许多 worker 节点可以针对各自负责的数据进行并行运算。

虽然这些基本概念的实现在不同的 Hadoop 系统上略有不同，但它们驱动了核心架构，并确保满足**容错性**、**可恢复性**、**一致性**和**可扩展性**的要求。这些要求又确保了 Hadoop 是一个数据分析处理符合预期的数据管理系统（传统上，这些分析处理是在关系数据库或科学数据仓库中执行的）。与数据仓库不同，Hadoop 能够在更经济的现成商业硬件上运行。因此，Hadoop 主要用于存储和计算大型异构数据集。快速分析和数据产品的原型设计有赖于这些存储在“数据湖泊”中（而不是数据仓库中）的数据集。

2.2 Hadoop架构

Hadoop 由两个主要组件组成：HDFS 和 YARN，它们实现了上一节讨论的分布式存储和计算的基本概念。HDFS（有时缩写为 DFS）是 Hadoop 的分布式文件系统，负责管理存储在集群中磁盘上的数据；YARN 则是集群资源管理器，将计算资源（worker 节点上的处理能力和内存）分配给希望执行分布式计算的应用程序。架构栈如图 2-1 所示。值得注意的是，原先的 MapReduce 应用程序和其他新的分布式计算应用程序，如图形处理引擎 Apache Giraph（http://giraph.apache.org）和内存计算平台 Apache Spark（http://spark.apache.org），现在基于 YARN 实现。

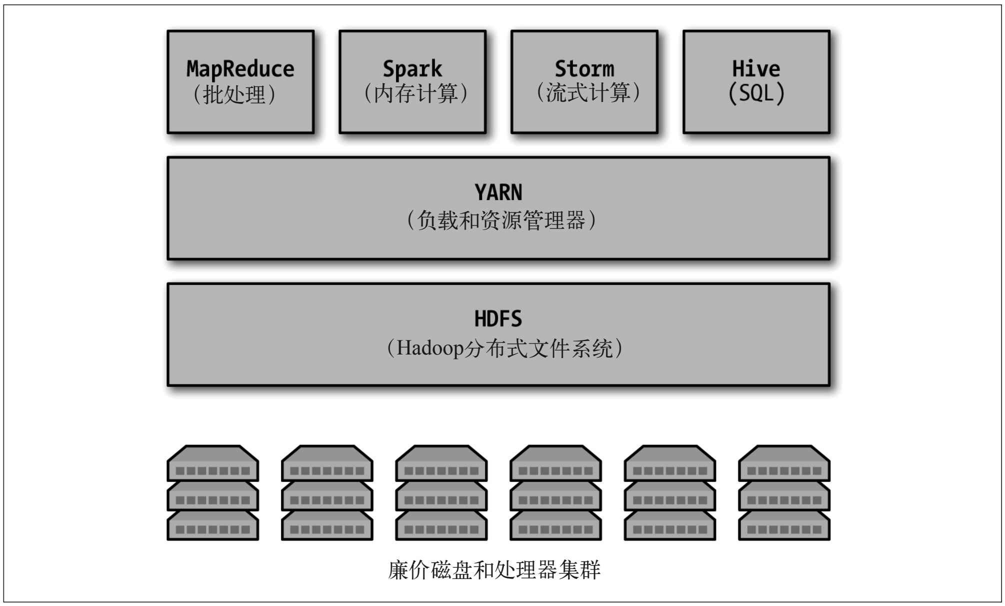


图 2-1：Hadoop 由 HDFS 和 YARN 构成

HDFS 和 YARN 协同工作，主要通过确保数据对于所需的计算是本地的，最大限度地减少集群中的网络流量。数据和任务的重复确保了容错性、可恢复性和一致性。此外，集群被集中管理，提供了可扩展性，还可将底层的集群编程细节抽象化。HDFS 和 YARN 共同构建了大数据应用程序的平台——也许不仅仅是一个平台，它们为大数据提供了一个操作系统。

和任何优秀的操作系统一样，HDFS 和 YARN 也很灵活。除 HDFS 之外的其他数据存储系统可以集成到 Hadoop 框架中，例如 Amazon S3 或 Cassandra。此外，数据存储系统也可以直接构建在 HDFS 之上，从而提供简单文件系统之外的功能。例如，HBase 是一个构建在 HDFS 之上的列式数据存储，它是利用分布式存储的一个先进的分析应用程序。在 Hadoop 的早期版本中，如果应用程序希望利用 Hadoop 集群上的分布式计算，它就必须将用户级实现转换为 MapReduce 作业。然而，YARN 现在支持对集群功能进行更丰富的抽象，这使得用于机器学习、图形分析、类 SQL 的数据查询，甚至流式数据服务的新数据处理应用

程序速度更快，且更容易实现。因此，围绕着 Hadoop，特别是 HDFS 和 YARN，丰富的工具和技术生态系统得以构建起来。

2.2.1 Hadoop集群

现在来问自己一个很有用的问题：**什么是集群**？到目前为止，我们一直在说 Hadoop 是一个以协调方式运行的机器集群。然而，Hadoop 并不是你必须购买或维护的硬件，它其实是运行在集群上的软件的名称，即 HDFS 和 YARN，它们由在一组计算机上运行的 6 种后台服务组成。

现在来逐一介绍它们。HDFS 和 YARN 提供了一个应用程序编程接口（Application Programming Interface，API），它使开发人员不必关注底层的集群管理细节。集群就是运行 HDFS 和 YARN 的一组计算机，每台计算机被称为一个节点。集群可以有一个节点，也可以有成千上万个节点，但是所有集群都是水平扩展的，这意味着在添加更多节点时，集群以线性方式提升容量和性能。

HDFS 和 YARN 由几个守护进程实现。守护进程是在后台运行并且不需要用户输入的软件。Hadoop 进程是服务，这意味着它们一直在集群节点上运行，接受输入并通过网络传递输出，这与 HTTP 服务器的工作原理类似。每个进程在自己的 Java 虚拟机（Java Virtual Machine，JVM）中运行，因此每个守护进程都有自己的系统资源分配，并由操作系统独立管理。集群中的每个节点都由其运行的一个或多个进程的类型标识。

master 节点
: 这些节点为 Hadoop 的 worker 节点提供协调服务，通常是用户访问集群的入口点。没有 master 节点，协调就不复存在，也就不可能进行分布式存储或计算。

worker 节点
: 集群中的大多数计算机都属于这类节点。worker 节点运行的服务从 master 节点接受任务——存储或检索数据、运行特定应用程序。worker 节点通过并行分析运行分布式计算。

HDFS 和 YARN 都有多个 master 服务，负责协调运行在各个 worker 节点上的 worker 服务。worker 节点实现 HDFS 和 YARN 的 worker 服务。HDFS 的 master 服务和 worker 服务如下所示。

NameNode（master 服务）
: 用于存储文件系统的目录树、文件元数据和集群中每个文件的位置。如果客户端想访问 HDFS，必须先通过从 NameNode 请求信息来查找相应的存储节点。

Secondary NameNode（master 服务）
: 代表 NameNode 执行内务任务并记录检查点。虽然它叫这个名字，但它并不是 NameNode 的备份。

DataNode（worker 服务）
: 用于存储和管理本地磁盘上的 HDFS 块，将各个数据存储的健康状况和状态报告给 NameNode。

从宏观上看，当从 HDFS 访问数据时，客户端应用程序必须先向 NameNode 发出请求，以在磁盘上定位数据。NameNode 将回复一个存储数据的 DataNode 列表，客户端必须直接从 DataNode 请求每个数据块。注意，NameNode 不存储数据，也不将数据从 DataNode 传递到客户端，而是像交警指挥交通一般，将客户端指向正确的 DataNode。

和 HDFS 类似，YARN 也有两个 master 服务和一个 worker 服务，如下所示。

ResourceManager（master 服务）
: 为应用程序分配和监视可用的集群资源（如内存和处理器核心这样的物理资源），处理集群上作业的调度。

ApplicationMaster（master 服务）
: 根据 ResourceManager 的调度，协调在集群上运行的特定应用程序。

NodeManager（worker 服务）
: 在单个节点上运行和管理处理任务，报告任务运行时的健康状况和状态。

与 HDFS 的工作方式类似，如果客户端希望执行作业，就必须先向 ResourceManager 请求资源，ResourceManager 会分配一个应用程序专用的 ApplicationMaster，它在作业的执行过程中会一直存在。ApplicationMaster 跟踪作业的执行，ResourceManager 则跟踪节点的状态，每个 NodeManager 创建容器并在其中执行任务。请注意，Hadoop 集群上也可能运行着其他进程（例如 JobHistory 服务器或 ZooKeeper 协调器），但这些服务是 Hadoop 集群中运行的主要软件。

主进程非常重要，所以它们通常在自己的节点上运行。因此，它们不会竞争资源，也不会带来瓶颈。但是在较小的集群中，所有的主守护进程可能都在一个节点上运行。以一个小型 Hadoop 集群的部署为例，它有六个节点，分别为两个 master 节点和四个 worker 节点，如图 2-2 所示。请注意，在较大的集群中，NameNode 和 Secondary NameNode 将驻留在不同的计算机上，从而避免竞争资源。因为集群是水平扩展的，所以集群的大小应该与预期的计算或数据存储能力呈正相关。一般来说，拥有 20~30 个 worker 节点和单个 master 节点的集群足以在几十太字节的数据集上同时运行多个作业。对于部署几百个节点的集群，每个 master 节点都拥有自己的计算机；而在拥有数千个节点的集群中，会有多个 master 节点被用于协调。

> 不一定要在集群上开发 MapReduce 作业；相反，大多数 Hadoop 开发人员通常在虚拟机中使用“伪分布式”开发环境。开发可以在一小部分数据样本上进行，而不必动用整个数据集。关于如何搭建伪分布式开发环境，请参见附录 A。

还有一种集群也很值得关注，那就是单节点集群。在“伪分布式模式”中，单个机器将运行所有 Hadoop 守护进程，就好像它是集群的一部分，但网络流量是通过本地环回网络接口流动的。这种模式虽然没有发挥出分布式架构的优势，但却是一种完美的开发模式，因为不必为管理几台机器而费心。Hadoop 开发人员通常使用伪分布式环境，该环境通常位于虚拟机内部，通过 SSH 连接虚拟机。Cloudera、Hortonworks 和其他流行的 Hadoop 发

行版提供了预先构建的虚拟机镜像，供你下载并立即使用。如果你想自己配置伪分布式节点，请参考附录 A。

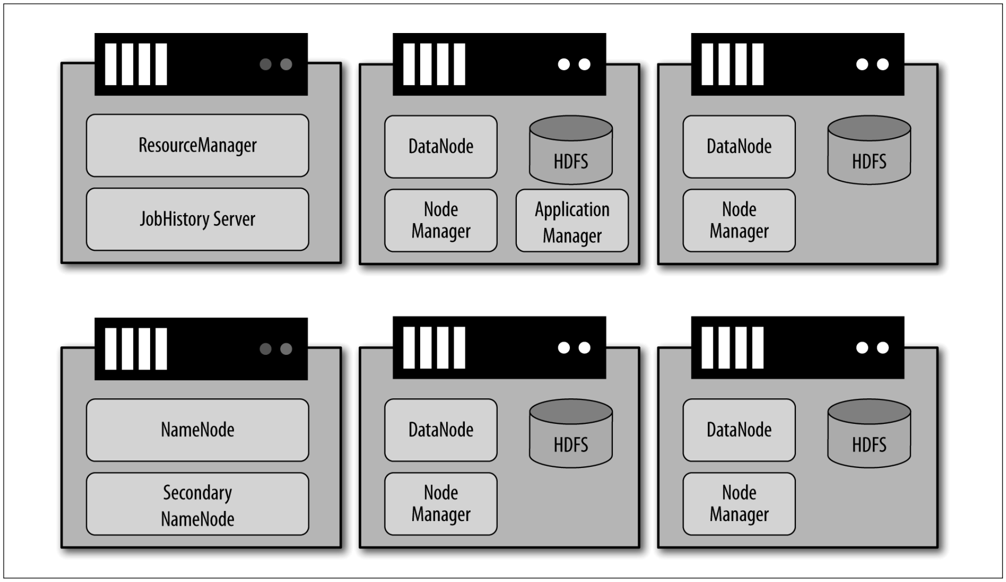


图 2-2：拥有两个 master 节点和四个 worker 节点的小型 Hadoop 集群，实现了全部六个主要的 Hadoop 服务

2.2.2 HDFS

通过将数据存储在由廉价、不可靠的计算机组成的集群中，HDFS 为大数据提供冗余存储，从而扩展单台计算机的可用存储容量。然而，由于分布式文件系统的网络特性，HDFS 比传统的文件系统更复杂。为了在最大程度上降低这种复杂性，HDFS 采用集中式存储架构。[2]

理论上，HDFS 是位于本机文件系统（如 ext4 或 xfs）之上的软件层。而实际上，Hadoop 一般化了存储层，可与本地文件系统和其他存储类型（如 Amazon S3）进行交互。HDFS 是分布式文件系统的旗舰，也是大多数编程场景所用的主要文件系统。它被设计用于存储非常大的文件，使用流访问数据，因此有一些注意事项。

- 与占用相同容量的数以亿计的小文件相比，HDFS 更适合处理数量适中但非常大的文件（例如几百万个 100MB 或更大的文件）。
- HDFS 采用 WORM 模式，即写一次，读多次（write once, read many）。它不允许随机写入或追加到文件。
- HDFS 针对文件的大型流式读取进行了优化，不采用随机读取或随机选择。

注 2：Ghemawat、Gobioff 和 Leung 于 2003 年发表的论文“The Google File System”（http://bit.ly/google-filesystem）率先提出这一理念。

因此，用 HDFS 来存储用于计算的原始输入数据、计算阶段之间的中间结果，以及整个作业的最终结果再合适不过了。但如果应用程序需要实时更新、交互式数据分析和基于记录的事务支持，它就不适合作为数据后端使用了。相反，通过写入一次并读取多次数据，HDFS 用户可以创建大量异构数据，以进行不同的计算和分析。这些数据存储有时被称为“数据湖泊”，因为它们以可恢复和容错的方式简单地保存关于已知问题的所有数据。本书稍后将讲解突破这些限制的变通方法。

1. 文件块

HDFS 文件分为多个块，块大小通常为 64MB 或 128MB。尽管这可在运行时配置，但是高性能系统通常将块大小设为 256MB。块大小是可以在 HDFS 中读取或写入的最小数据量，类似于单个磁盘文件系统上的块大小。但是与单个磁盘上的块不同，小于块大小的文件不占用实际文件系统上一个完整块的空间。这意味着为了实现最佳性能，Hadoop 更喜欢分解成小块的大文件，能将许多较小的文件合并成一个大文件就很好。但是如果 HDFS 上存储了许多小文件，就不是每个文件都能让总可用磁盘空间减少 128MB 了。

块能将运行中的非常大的文件拆分并将其分发到许多计算机上。来自同一文件的不同块将被存储在不同的计算机上，以提供更高效的分布式处理。事实上，在任务和数据块之间存在一对一的关系。

此外，块将跨 DataNode 复制。默认情况下，块将复制三份，但也可在运行时配置。因此，每个块都将分布在三台计算机和三块磁盘上。即使两个节点都发生了故障，数据也不会丢失。请注意，这意味着集群中的潜在数据存储容量仅为可用磁盘空间的三分之一，但因为磁盘通常非常便宜，所以这在大多数数据应用程序中都不成问题。

2. 数据管理

主 NameNode 记录组成文件的块和这些块所在的位置。NameNode 与 DataNode（集群中实际保存块的进程）进行通信。与每个文件相关联的元数据被存储在 NameNode 的 master 节点的内存中，以便进行快速查找。如果 NameNode 停止或发生故障，整个集群都将无法访问！

Secondary NameNode 不是 NameNode 的备份，而是代表 NameNode 执行内务任务，包括（特别是）定期将当前数据空间的快照与编辑日志合并，以确保编辑日志不会过大。编辑日志用于确保数据的一致性，防止数据丢失。如果 NameNode 发生故障，就可以用合并后的记录重建 DataNode 的状态。

当客户端应用程序想要读取文件时，它首先从 NameNode 请求元数据，以定位组成文件的块以及存储块的 DataNode 的位置。然后，应用程序直接与 DataNode 通信以读取数据。因此，NameNode 仅仅扮演着日志或查找表的角色，而不是同时读取的瓶颈。

2.2.3 YARN

虽然原始版本的 Hadoop（Hadoop 1）普及了 MapReduce，并使大众接触到了大规模的分布式处理，但它只在 HDFS 上提供 MapReduce。这是因为在 Hadoop 1 中，MapReduce 作业 / 工作负载管理功能与集群 / 资源管理功能高度耦合。因此，其他处理模型或应用程序无法将集群基础设施用于其他分布式工作负载。

虽然 MapReduce 在批量处理大规模工作负载时非常高效，但是它是 I/O 密集型的，并且由于 HDFS 和 MapReduce 的面向批处理性质，它在支持交互式分析、图形处理、机器学习和其他内存密集型算法时面临明显的限制。虽然已经为这些特定的场景开发了其他分布式处理引擎，但是 Hadoop 1 专注于 MapReduce 的本质决定了它不可能改变同一集群的用途，转而去支持这些分布式工作负载。

Hadoop 2 通过引入 YARN 突破了这些限制。YARN 将工作负载管理与资源管理分离，以便多个应用程序可以共享一个集中的公共资源管理服务。通过在 YARN 中提供通用的作业和资源管理能力，Hadoop 不再是一个仅仅专注于 MapReduce 的框架，而成为了一个完整、多应用程序的大数据操作系统。

2.3　使用分布式文件系统

请记住，HDFS 实际上是一个分布式远程文件系统。它与 POSIX 文件系统的相似性很容易误导我们，特别是文件系统查找的所有请求都发送到 NameNode。NameNode 能够快速响应查找类型的请求。一旦你开始访问文件，速度会很快慢下来，因为组成请求文件的各个块都必须通过网络传输到客户端。还要记住，因为块在 HDFS 上有多个副本，所以 HDFS 中的可用磁盘空间实际上比硬件提供的可用磁盘空间少。



以下示例提供的命令和环境变量可能与你使用的 Hadoop 版本或系统不同。在大多数情况下，这些差异应该很容易理解。本书假设你使用和附录 A 描述的伪分布式节点一样的设置。

在大多数情况下，与 HDFS 的交互是通过命令行接口进行的，使用过 Unix 或 Linux 上的 POSIX 接口的用户都很熟悉这种方式。此外，HDFS 还有一个 HTTP 接口和一个用 Java 编写的可编程接口。但是因为开发人员最熟悉命令行接口，所以本书将从命令行接口入手。

在本节中，我们将通过命令行完成与分布式文件系统的基本交互。假设执行这些命令的是可以连接到远程 Hadoop 集群的客户端，或本地主机上运行着伪分布式集群的客户端。此外，本书也假设 `hadoop` 命令和 $HADOOP_HOME/bin 中的其他工具位于系统路径上，并且可以被操作系统找到。

2.3.1　基本的文件系统操作

用户可以进行所有常规的文件系统操作，例如创建目录，移动、删除和复制文件，列出目录内容，修改集群上文件的权限。要查看 `fs` 命令下可用的命令，键入：

```
hostname $ hadoop fs -help
Usage: hadoop fs [generic options]
...
```

如你所见，许多与文件系统交互的常见命令都可用，通过 `hadoop fs` 命令的参数指定——就像 Java 风格的标志参数，即在命令后加短横线（`-`）。这些命令的辅助标志或选项另外使用由空格分隔的 Java 样式参数指定，跟在初始命令之后。请注意，指定此类选

项的顺序很重要。

开始动手吧。先使用 put 或 copyFromLocal 命令从本地文件系统复制一些数据到远程（分布式）文件系统。这两个命令是相同的，都能将文件写入分布式文件系统，而不删除本地副本。moveFromLocal 命令的功能类似，但会在文件成功传输到分布式文件系统后，将本地副本删除。

在存放本书代码和资源的 GitHub 仓库（https://github.com/bbengfort/hadoop-fundamentals）中，有一个 /data 目录，其中有一个名为 shakespeare.txt 的文件，包含了莎士比亚作品全集。将此文件下载到你的本地工作目录。下载后，将文件复制到分布式文件系统，如下所示：

```
hostname $  hadoop fs -copyFromLocal shakespeare.txt shakespeare.txt
```

本示例调用 Hadoop 的 shell 命令 copyFromLocal，并带有 <src> 和 <dst> 两个参数，它们都被指定为 shakespeare.txt 文件的相对路径。整个过程为：copyFromLocal 命令在当前工作目录中搜索 shakespeare.txt 文件，从 NameNode 请求有关该路径的信息，然后直接与 DataNode 通信以传送文件，将其复制到 HDFS 上的 /user/analyst/shakespeare.txt 路径。因为莎士比亚作品全集小于 64MB，所以它不会被分成块。但是请注意，在本地计算机以及远程 HDFS 系统上，都必须考虑相对路径和绝对路径。以上命令的全写形式如下：

```
hostname $ hadoop fs -put /home/analyst/shakespeare.txt \
           hdfs://localhost/user/analyst/shakespeare.txt
```

你可能注意到了，HDFS 上的主目录和 POSIX 系统上的主目录很像。/user/analyst/ 目录就是主目录——analyst 用户的主目录。远程文件系统的相对路径将用户的 HDFS 主目录视为当前工作目录。事实上，HDFS 文件和目录的权限模型和 POSIX 的非常像。为了更好地管理 HDFS 文件系统，像在本地文件系统上一样创建目录的分层树：

```
hostname $ hadoop fs -mkdir corpora
```

要列出远程主目录的内容，可以使用 ls 命令：

```
hostname $ hadoop fs -ls .
drwxr-xr-x  -  analyst analyst       0 2015-05-04 17:58 corpora
-rw-r--r--  3  analyst analyst 8877968 2015-05-04 17:52 shakespeare.txt
```

HDFS 文件列举命令与具备一些 HDFS 特定特征的 Unix ls -l 命令很像。但当此命令不带任何参数时，则会提供用户的 HDFS 主目录的列表。第 1 列显示文件的权限模式，第 2 列是文件的副本数（默认情况下，副本数为 3）。请注意，目录不会被复制，因此本例中的此列是短横线（-）。其后依次是用户、组、以字节为单位的文件大小（目录为零）、最后一次修改的日期和时间，以及文件名。

其他基本的文件操作（如 mv、cp 和 rm）在远程文件系统上的工作方式和预想的一样。但是，删除目录时不使用 rmdir 命令，而是使用 rm -R 递归删除目录及其包含的所有文件。

将文件从分布式文件系统读取和移动到本地文件系统时应小心处理，因为分布式文件系统维护的文件非常大。有些情况下，用户需要详细检查文件，特别是 MapReduce 作业的结果生成的输出文件。这些文件通常不被读取到标准输出流，而是使用管道传输到其他程序，如 less 或 more。

如需读取文件的内容，可使用 cat 命令，然后将输出通过管道传递给 less 以查看远程文件的内容：

```
hostname $ hadoop fs -cat shakespeare.txt | less
```

> 当使用 less 时，可以通过方向键导航文件，键入 q 退出并退回到终端。

还可以使用 tail 命令仅检查文件的最后 1000 字节：

```
hostname $ hadoop fs -tail shakespeare.txt | less
```

没有类似 hadoop fs -head 的命令可以用来检查文件的前 1000 字节。不过，可以使用 hadoop fs -cat 并通过管道将文件内容传输到本地 shell 的 head 命令。这种做法很高效，因为 head 命令在读取整个文件之前就终止了远程流。但是以这种方式使用 shell 的 tail 会降低效率，因为在计算输出之前，所有数据都必须从远程文件系统流式传输到本地文件系统。相反，hadoop fs -tail 命令在远程文件中寻址到正确位置，仅通过网络返回所需的数据。

若想将整个文件从分布式文件系统传输到本地文件系统，可以使用 get 或 copyToLocal，这两个命令是相同的。与之类似，也可以使用 moveToLocal 命令，但它会将该文件从分布式文件系统中删除。get merge 命令复制符合给定模式或指定目录下的所有文件，并将其合并成本机的单个文件。如果远程系统上的文件较大，可以在管道传输的时候使用压缩工具：

```
hostname $ hadoop fs -get shakespeare.txt ./shakespeare.from-remote.txt
```

比较后可以发现，原始的 shakespeare.txt 文件与 shakespeare.from-remote.txt 文件相同。希望这些示例充分展现了 hadoop fs 命令是一个功能齐全的 HDFS 命令行接口，而且经常用于开发分析作业。表 2-1 展示了 hadoop fs 提供的其他命令，它们也都很有用。

表2-1：其他有用的命令

命令	输出
hadoop fs -help *<cmd>*	提供特定于 *<cmd>* 的信息和标志
hadoop fs -test *<path>*	回答各种关于 *<path>* 的问题（例如它是否存在、是否是目录、是否是文件，等等）
hadoop fs -count *<path>*	统计符合指定文件模式的路径所包含的目录数、文件数和字节数
hadoop fs -du -h *<path>*	显示符合指定文件模式的文件使用的空间字节数
hadoop fs -stat *<path>*	打印 *<path>* 路径的文件 / 目录的统计数据
hadoop fs -text *<path>*	获取一个源文件并以文本格式输出文件内容，目前支持 Zip、TextRecord InputStream 和 Avro 文件

2.3.2 HDFS文件权限

如前所述，HDFS 的文件权限与 POSIX 类似。权限分三种类型：读（r）、写（w）和执行（x）。这些权限定义了所有者、组和任何其他系统用户的访问级别。对于目录来说，执行

权限允许访问目录的内容，但是 HDFS 上文件的执行权限被忽略了。在 HDFS 中，读写权限指定谁可以访问数据，以及谁可以追加文件内容。

目录列举命令 `ls` 能显示权限信息。每个模式有 10 个槽位。第 1 个槽位是 `d`，表示“是目录，否则是文件（`-`）”。接下来的槽位每 3 个为一组，分别表示所有者、组和其他用户的 `rwx` 权限。有几个 HDFS 的 shell 命令能管理文件和目录的权限，即我们熟悉的 `chmod`、`chgrp` 和 `chown` 命令：

```
hostname $ hadoop fs –chmod 664 shakespeare.txt
```

在上例中，`chmod` 命令将 shakespeare.txt 的权限更改为 `-rw-rw-r--`。664 是为权限三元组设置的标志的八进制表示。6 的二进制数为 110，这意味着设置了读和写的标志，但没有设置执行标志；完全允许是 7，即二进制数 111；只读是 4，即二进制数 100。`chgrp` 和 `chown` 命令分别更改分布式文件系统上文件的组和所有者。

设置 HDFS 文件权限时需要注意，客户端的身份是由跨 HDFS 运行的进程的用户名和组确定的，这意味着远程客户端可以在系统上创建任意用户。因此，这些权限只用于防止数据意外丢失，以及在已知用户之间共享文件系统资源，而不作为安全机制使用。

2.3.3 其他HDFS接口

软件开发人员可以通过 Java API 访问 HDFS 的编程接口，并且所有将数据采集到 Hadoop 集群的过程都应考虑使用该 API。还有一些可以将 HDFS 与其他文件系统或网络协议（如 FTP 或 Amazon S3）集成的工具。第 6 章将更加关注数据管理问题，以及如何从各种源获取数据并将其存储到 HDFS。

HDFS 也有 HTTP 接口，可用于集群文件系统的常规管理以及使用 Python 访问 HDFS。HDFS 的 HTTP 接口主要有两种：一是通过处理 HTTP 请求的 HDFS 守护进程直接访问，二是由代理服务器暴露 HTTP 接口，然后代表客户端使用 Java API 直接访问 HDFS。代理包括 HttpFS、Hoop 和 WebHDFS，它们都支持 RESTful 网络访问 Hadoop 集群，这很容易用 Python 实现。

通过在端口 50070 上运行的 HTTP 服务器，NameNode 也提供对 HDFS 的直接只读 HTTP 访问。如果以伪分布式模式运行，只需打开浏览器并访问 http://127.0.0.1:50070；否则，就使用集群上 NameNode 的主机名。NameNode Web UI 提供了集群状态的总览情况，包括可用和已用的存储容量、活动和死亡的 DataNode 数量，以及副本数不足的块的警告信息。

通过使用 Utilities 下拉选项卡中的搜索和导航工具，NameNode 还允许用户浏览文件系统。文件元信息的列举方式类似于命令行接口，特定文件还允许下载。通过访问 DataNode 主机的 50075 端口，可以直接浏览 DataNode 上的信息，所有活动的 DataNode 都会在 NameNode 的 HTTP 站点上列出。

默认情况下，HTTP 的直接访问接口是只读的。为了提供对 HDFS 集群的写访问，必须使用 WebHDFS 等代理。WebHDFS 通过使用 Kerberos 进行身份验证，从而保护集群。如何访问 Hadoop 上的安全资源主要取决于集群的特定配置，以及是否使用了任何第三方管理工具。Hadoop 用于运行在完全托管且不暴露于外部世界的内部集群上，因此 Hadoop 的安

全性不像其平台本身那么成熟——这也是 Hadoop 继续发展需要考虑的主要问题之一。

2.4 使用分布式计算

到目前为止，你应该已经能通过命令行轻松与集群（甚至是伪分布式集群）交互了。大多数数据科学家和软件开发人员应该对上一节展示的文件系统命令很熟悉。除了在大型数据集的管理和集群中的网络通信方面有一些差异之外，HDFS 可以被很轻松地集成到当前的操作工作流中。本书接下来的内容将主要关注驻留在 HDFS 上的数据的管理和计算。为了实现这一点，我们需要对分布式计算及其要求有基本了解。

虽然 YARN 已经使 Hadoop 成为一个通用的分布式计算平台，但 MapReduce（通常缩写为 MR）是 Hadoop 的第一个计算框架。YARN 让非 MapReduce 框架（如 Spark、Tez 和 Storm，仅举几例）可以与原先的 MapReduce 应用程序一起在 Hadoop 集群上运行。但是，对于大多数 Hadoop 用户来说，MapReduce 仍然是许多应用程序和分析的主要框架。此外，对 MapReduce 的工作原理有所了解，能帮助我们更深刻地理解分布式分析，还可以讨论其他平台是如何工作的，因为 MapReduce 的理论基础与其他框架是相同的。

本节将探究 MapReduce 编程范式的基本原理，并讨论为何这些函数式编程结构会成为分布式系统的理想选择。我们将通过两个简单的分析示例（单词计数和共同好友）演示 MapReduce 如何工作，这两个示例通常用于演示分布式环境中的计算。最后，本节将描述如何在 Hadoop 集群上实现 MapReduce 应用程序，同时演示如何提交和管理示例 MapReduce 作业，并通过 Hadoop 命令行接口获取输出。

2.4.1 MapReduce：函数式编程模型

当人们提到 MapReduce 时，通常指的是分布式编程模型。[3]MapReduce 是一个简单但功能强大的计算框架，专门用于在集中管理的机器集群上进行容错的分布式计算。它采用了先天可并行的“函数式”编程风格来实现，允许多个独立任务在本地数据块上执行函数，并在处理后聚合结果。

函数式编程是一种编程风格，它确保每一个计算单元以无状态的方式被评估。这意味着函数仅依赖于输入，并且是封闭且不共享状态的。通过将一个函数的输出作为另一个完全独立的函数的输入，函数之间实现数据传输。这些特征使得函数式编程非常适合分布式的大数据计算系统，因为它允许我们将计算移动到任何有数据输入的节点，并保证得到的结果相同。因为函数是无状态的，且仅仅依赖于它们的输入，所以多台机器上的多个函数可以独立处理一小部分数据。通过将一些函数的输出策略性地链接到其他函数的输入，可以实现整个数据集的最终计算。

负责分发任务和聚合结果的两类函数分别被称为 `map` 和 `reduce`。这些函数运算的输入和输出数据不是简单的列表或值的集合，MapReduce 其实是利用键值对来协调计算。因此，Python 中 `map` 和 `reduce` 函数的伪代码如下所示：

注 3：分布式编程模型由 Google 设计，Jeffrey Dean 和 Sanjay Ghemawat 后来在论文“MapReduce: Simplified Data Processing on Large Clusters”（http://bit.ly/google-mapreduce-paper）中介绍了它。

```
def map(key, value):
    # 执行处理
    return (intermed_key, intermed_value)

def reduce(intermed_key, values):
    # 执行处理
    return (key, output)
```

map 函数以一系列键值对作为输入，然后在每个键值对上进行单独运算。以上伪代码按照它在 MapReduce 的 Java API 中的表示表达了这一概念：一个接受两个参数（一个键和一个值）的函数。在对输入数据执行了一些分析或变换之后，map 函数输出零个或多个键值对，在以上伪代码中表示为单个元组。整个过程将 map 函数应用于输入列表以创建新的输出列表，如图 2-3 所示。

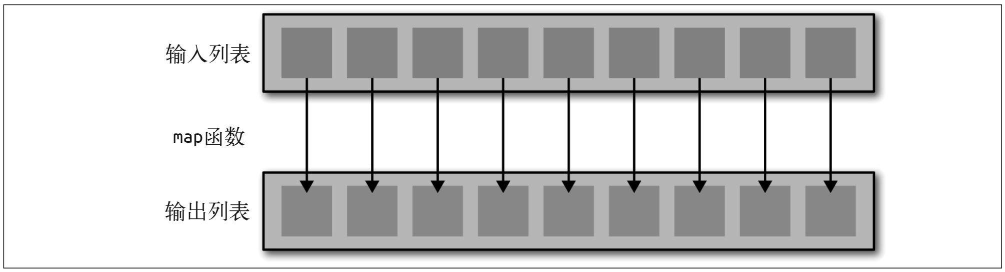


图 2-3：map 函数以一系列键值对作为输入，然后在每个键值对上进行单独运算，输出零个或多个键值对

通常来讲，map 操作将进行核心分析或处理，因为这个函数能查看数据集中的每个元素。想想如何在 map 中实现过滤器：测试每个键值对，确定它是否属于最终数据集；如果是则发出，否则就忽略。在 map 阶段之后，所有发出的键值对将按照键来分组，然后根据键被用于各个 reduce 函数的输入。如图 2-4 所示，reduce 函数被应用于一个输入列表以输出单个聚合值。

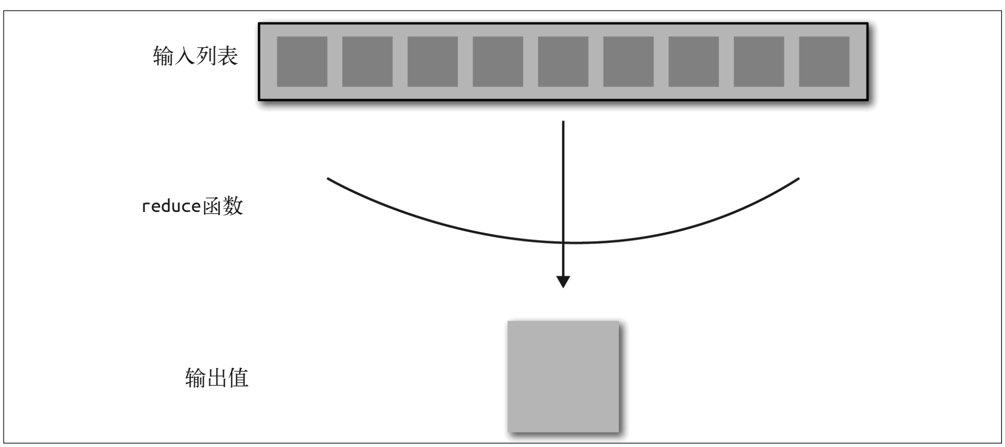


图 2-4：reduce 函数以一个键和一个值列表作为输入，通常通过聚合操作在整个值列表上进行运算，输出零个或多个键值对

如伪代码（与 MapReduce 的 Java API 有些类似）所示，`reduce` 函数是一个有两个参数的函数：一个键（在伪代码中是 `intermed_key`），以及与之相关联的迭代器或值列表（`values`）。reducer 对值列表执行最终处理，通常是组合或聚合，然后输出零个或多个键值对。reducer 旨在聚合从 map 阶段输出的大量值，以便将大量数据转换为更小、更易于管理的概要数据。但它的用处可不止如此。

2.4.2 MapReduce：集群上的实现

因为 mapper 对任意列表的每个元素应用相同的函数，所以非常适合被分发到集群的节点上。每个节点获得一个 mapper 操作的副本，并且将 mapper 应用于存储在本地 HDFS 数据节点的数据块中的键值对上。独立处理数据的 mapper 的数量不定，只受集群上可用的处理器的数量限制。因为它们是无状态的，所以进程之间不需要（或不可能）进行网络通信。又因为 mapper 具有确定性，它们的输出不依赖于输入值以外的内容，所以可以在另一个节点上重试失败的 mapper。

reducer 需要根据键获取 mapper 的输出作为输入，因此 reducer 的计算也可以被分发出去。如此一来，reduce 运算的数量最多可能与 mapper 输出中的可用键一样多。可以预见，每个 reducer 都应该能看到某个独一无二的键的**所有**值。为了满足该要求，需要 shuffle 和 sort 操作来协调 map 和 reduce 阶段，使 reducer 的输入按键分组并排序。shuffle 和 sort 将 map 阶段的键空间分区，以便将特定键空间分配给特定 reducer。总体来说，MapReduce 的阶段如图 2-5 所示。

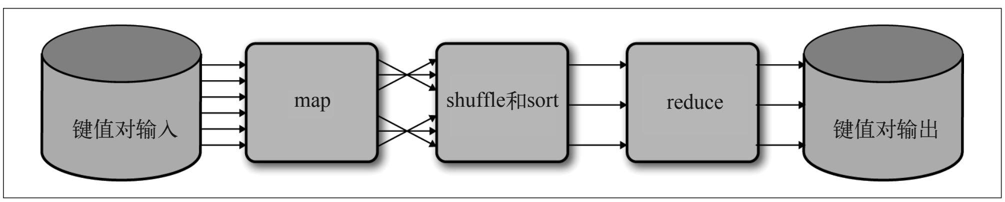


图 2-5：总体来说，MapReduce 是一个分阶段框架，其中的 map 阶段和 reduce 阶段通过中间的 shuffle 和 sort 协调

图 2-5 中涉及的阶段如下所示。

阶段 1

HDFS 的本地数据以键值对的形式被加载到一个映射过程。

阶段 2

mapper 输出零个或多个键值对，将计算所得的值映射到一个特定的键上。

阶段 3

基于键对这些键值对进行 sort 和 shuffle 操作，然后将它们传递给 reducer，使 reducer 获得键的所有值。

阶段 4

reducer 输出零个或多个最终的键值对，即输出（归约 map 的结果）。

在大多数情况下，数据工程师只需要关注 MapReduce 的这一宽泛描述便可以实现分析应用程序。但在集群上执行 MapReduce 时，还需要其他详细信息。例如，键值对是如何定义的？对键空间进行正确分区需要什么？你也可能需要进行改进和优化，如加入 combiner 和其他中间阶段，从而使简单的作业仅通过较少的计算资源就能完成。在拥有几个节点的集群上，MapReduce 流水线的数据流细节（尽管这超出了本书的范畴）如图 2-6 所示。

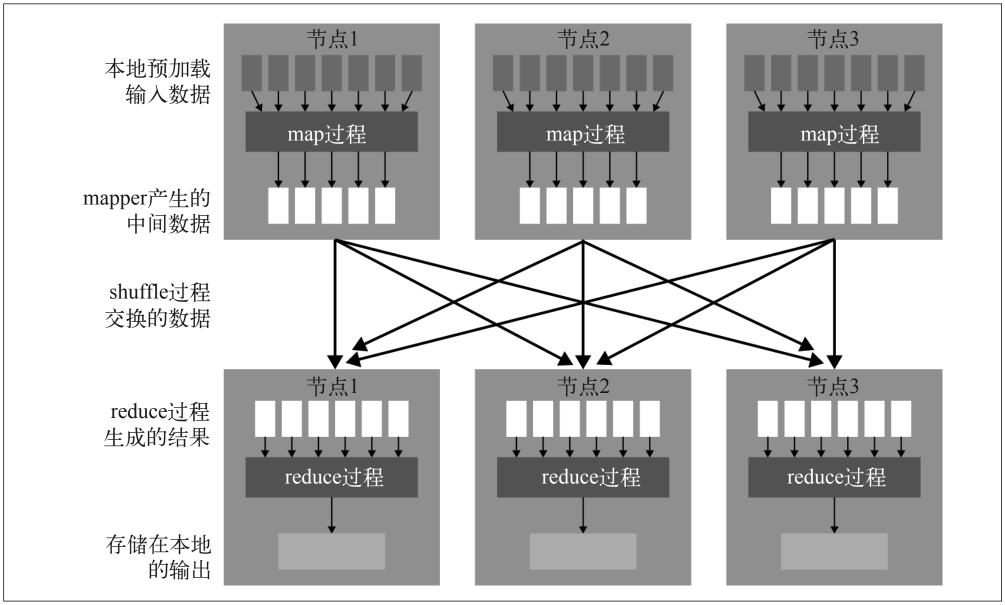


图 2-6：在拥有几个节点的集群上执行的 MapReduce 作业的数据流

在集群执行上下文中，map 任务被分配给集群中的一个或多个节点，这些节点包含指定为 map 操作输入的本地数据块。块存储在 HDFS 中，并由 `InputFormat` 类拆分为更小的块，该类定义了如何将数据呈现给 map。例如，给定文本数据，键可以是文件标识符和行号，值可以是行内容的字符串。`RecordReader` 将每个键值对呈现给用户提供的 map 操作，map 再输出一个或多个中间键值对。这时一般会使用 combiner 进行优化，它聚合单个 mapper 的输出，和 reducer 的工作原理类似，但是不涉及整个键空间。这项预备工作减少了 reducer 的工作量，能提高其性能。

中间阶段的键从 map 过程被拉到 partitioner，partitioner 决定如何将键分配给 reducer。通常假定键空间是均匀分布的，因此散列函数用于在 reducer 之间均匀地划分键。partitioner 还对键值对进行排序，以便实现完整的“shuffle 和 sort”阶段。最后，reducer 开始工作，为每个键生成数据迭代器并执行 reduce 操作，如聚合。输出的键值对将使用 `OutputFormat` 类写回到 HDFS。

在 MapReduce 集群执行上下文中，也有其他许多管理大规模作业的工具。仅举几个例子，`Counter` 和 `Reporter` 对象用于作业跟踪和评估，缓存用于在处理期间提供辅助数据。高级框架（如 Pig 或 Hive）通常都实现了这些工具，供开发人员使用。但在第 3 章中，我们将看到如何使用 Python 和 Hadoop Streaming 来实现这些功能。

MapReduce示例

为了演示数据流经 map 和 reduce 的过程，我们将演示两个具体的例子：**单词计数**和**共同好友**。这两个应用程序虽然简单，但却能演示分布式系统内的数据流动过程。特别是单词计数，因为它经常被用于演示分布式计算任务，所以通常被称为大数据界的“Hello, World”。因为单词计数和共同好友属于“易并行”问题，所以它们不仅能帮助我们理解 MapReduce，还能指出应用程序的设计是否存在根本缺陷。

单词计数应用程序以一个或多个文本文件作为输入，生成一份单词及其频率的列表。具体来说，因为 Hadoop 使用键值对，所以输入的键是文件 ID 和行号，输入的值是字符串，而输出的键是单词，输出的值是一个整数。我们立刻就会发现，这可以采用多种方式并行化。首先，每个 mapper 可以处理单个文档；如果文档非常大，mapper 可以处理单个文档中的多个块——map 操作不关心单词的上下文，它只统计作为输入的单词的个数。同理，可以用多个 reducer 同时处理不同的键，因为输出键是一个单词。以下 Python 伪代码展示了如何实现此算法：

```
# emit是一个执行Hadoop I/O的函数 ❶

def map(dockey, line):
    for word in Line.split():
        emit(word, 1)

def reduce(word, values):
    count = sum(value for value in values)
    emit(word, count)
```

❶ 从参数的角度考虑，`emit` 是一个执行 Hadoop I/O 的函数；也就是说，它将其参数发送到 MapReduce 流水线的下一个阶段，类似于 Python 中的 `yield` 函数。

在图 2-7 中有两个文档，包含两个简单句子。`map` 函数将接收文本的某个唯一 ID，以及该文档内容的字符串。它通过空格和标点符号分割值（获取所有单词），并将每个单词作为中间键、值为 1 发出——因为 mapper 已经看到该单词出现了一次。每个 mapper 的数据如下所示：

```
# WordCount mapper的输入

(27183, "The fast cat wears no hat.")
(31416, "The cat in the hat ran fast.")

# mapper 1的输出

("The", 1), ("fast", 1), ("cat", 1), ("wears", 1),
("no", 1), ("hat", 1),(".", 1)

# mapper 2的输出

("The", 1), ("cat", 1), ("in", 1), ("the", 1),
("hat", 1), ("ran", 1),("fast", 1),(".", 1)
```

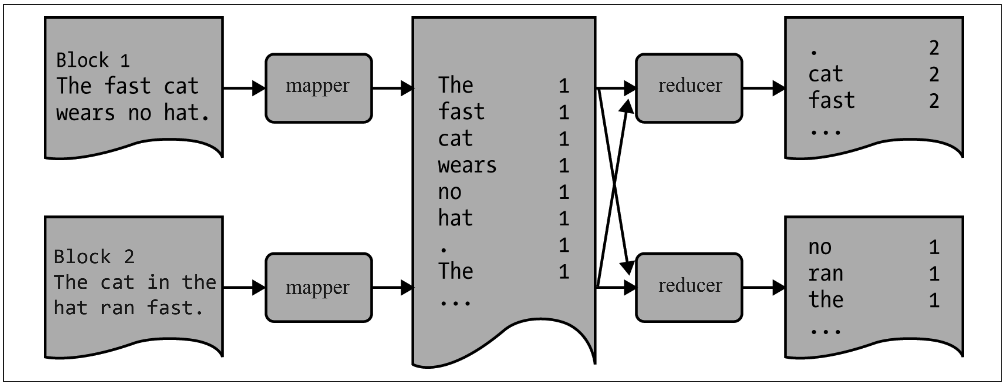


图 2-7：在集群上执行的有两个 mapper 和两个 reducer 的单词计数作业的数据流

这些数据被传递到 shuffle 阶段和 sort 阶段，键（单词）被分组并排序，然后发送到适当的 reducer。每个 reducer 接收以单词作为键、一串数字 1 作为值的输入。为了获得计数，它简单地将这些数字 1 相加，并将单词作为键、计数作为值发出。示例中的输入和输出的数据如下所示：

```
# WordCount reducer的输入
# 该数据由shuffle和sort计算

(".", [1, 1])
("cat", [1, 1])
("fast", [1, 1])
("hat", [1, 1])
("in", [1])
("no", [1])
("ran", [1])
("the", [1])
("wears", [1])
("The", [1, 1])

# 所有WordCount reducer的输出

(".", 2)
("cat", 2)
("fast", 2)
("hat", 2)
("in", 1)
("no", 1)
("ran", 1)
("the", 1)
("wears", 1)
("The", 2)
```

这种算法看似简单，但是它稍微复杂一点的实现常被用于文本处理。想象一下如何计算《纽约时报》或 Google 图书语料库中最常出现的单词——这肯定需要某种大数据技术。使用 *n*-gram 语言模型可以对同时出现的单词进行计数，以查看一起出现的两个单词之间是否存在统计意义，如 white house（白宫）或 baseball bat（棒球棒）。此外，了解数据如何从

输入源通过 map 操作流到 reduce 操作再产生输出，对于在分布式环境中开发分析过程和数据工程任务至关重要。

接下来学习一个稍微复杂些的例子，以确保 MapReduce 有意义。共同好友任务的目标是通过分析社交网络，查看用户间有哪些共同好友。这既是进行下游分析（例如“你可能认识的人”推荐）的第一步，也是实现只允许你与朋友及朋友的朋友分享内容的社交网络的一个关键部分。给定输入数据源，其中键是用户的名称，值是用逗号分隔的朋友列表，以下 Python 伪代码演示如何执行此计算：

```
def map(person, friends):
    for friend in friends.split(","):
        pair = sort([person, friend])
        emit(pair, friends)

def reduce(pair, friends):
    shared = set(friends[0])
    shared = shared.intersection(friends[1])
    emit(pair, shared)
```

mapper 从初始数据集创建中间键空间，其中包含所有可能存在的 `(friend, friend)` 元组。因为值是好友列表，所以可以针对每个关系分析数据集。此外，请注意该关系对已经被排序，这确保了 `("Mike", "Linda")` 和 `("Linda", "Mike")` 在 reducer 聚合时是相同的键。输入和 mapper 输出如下所示：

```
# 输入(键 → 值)
Allen → Betty, Chris, David
Betty → Allen, Chris, David, Ellen
Chris → Allen, Betty, David, Ellen
David → Allen, Betty, Chris, Ellen
Ellen → Betty, Chris, David

# mapper 1的输出
(Allen, Betty) → (Betty, Chris, David)
(Allen, Chris) → (Betty, Chris, David)
(Allen, David) → (Betty, Chris, David)

# mapper 2的输出
(Allen, Betty) → (Allen, Chris, David, Ellen)
(Betty, Chris) → (Allen, Chris, David, Ellen)
(Betty, David) → (Allen, Chris, David, Ellen)
(Betty, Ellen) → (Allen, Chris, David, Ellen)

# mapper 3的输出
(Allen, David) → (Allen, Chris, David, Ellen)
(Betty, David) → (Allen, Chris, David, Ellen)
(Chris, David) → (Allen, Chris, David, Ellen)
(David, Ellen) → (Allen, Chris, David, Ellen)

# mapper 4的输出
(Betty, Ellen) → (Betty, Chris, David)
(Chris, Ellen) → (Betty, Chris, David)
(David, Ellen) → (Betty, Chris, David)
```

针对数据集中存在的每对朋友关系，reducer 确定可以看到两个好友列表，每个好友列表对应键中的一个用户。因此，为了执行最终聚合，reducer 先简单地将这些列表转换成集合，并求两者间的交集，即共同好友；然后，发射该交集，其中包含按字母顺序排列的关系元组和相关的好友。请注意，reducer 可以简单地为关系中的每个人发射结果，这可能对其他应用程序的下游数据加载有帮助。流入 reducer 的数据如下所示：

```
# 经过shuffle和sort之后,reducer的输入

(Allen, Betty) → (A C D E) (B C D)
(Allen, Chris) → (A B D E) (B C D)
(Allen, David) → (A B C E) (B C D)
(Betty, Chris) → (A B D E) (A C D E)
(Betty, David) → (A B C E) (A C D E)
(Betty, Ellen) → (A C D E) (B C D)
(Chris, David) → (A B C E) (A B D E)
(Chris, Ellen) → (A B D E) (B C D)
(David, Ellen) → (A B C E) (B C D)

# reduce之后

(Allen, Betty) → (Chris, David)
(Allen, Chris) → (Betty, David)
(Allen, David) → (Betty, Chris)
(Betty, Chris) → (Allen, David, Ellen)
(Betty, David) → (Allen, Chris, Ellen)
(Betty, Ellen) → (Chris, David)
(Chris, David) → (Allen, Betty, Ellen)
(Chris, Ellen) → (Betty, David)
(David, Ellen) → (Betty, Chris)
```

本节中的具体示例（单词计数和共同好友）展示了数据流经 MapReduce 作业的过程，并给出了如何开发这种作业的思路——从想象 map 阶段和 reduce 阶段的数据流动过程开始就不错。确定需要输入和输出的键也有助于指导流水线的每个阶段应该做什么。

2.4.3 不止一个MapReduce：作业链

将常规解决问题的工作流稍作转变以满足 `map` 函数和 `reduce` 函数的无状态运算和交互后，就可以用 MapReduce 轻松实现许多算法或数据处理任务。但是单个 MapReduce 作业无法实现更复杂的算法和分析。例如，许多机器学习或预测分析技术需要优化，即最小化误差的迭代过程。MapReduce 不支持通过单个 map 或 reduce 进行迭代。

看来有必要对这些术语进行一番讨论。在 MapReduce 中，作业实际上指的是完整的应用程序（application 或 program），即对所有输入数据执行 `map` 函数和 `reduce` 函数的完整过程。复杂的分析作业通常由许多内部任务组成，其中的任务是指对一个数据块执行单个 `map` 运算或 `reduce` 运算的过程。因为有许多 worker 节点在同时执行类似的任务，所以一些数据处理工作流可以运行“只有 map”或“只有 reduce”的作业。例如，分箱方法可以利用内置的 partitioner 将类似的数据分在一组。分箱后的数据可以用于下游的其他 MapReduce 作业，执行频率分析或计算概率分布。

事实上，更复杂的应用程序是通过被称为“作业链”的过程，使用多个 MapReduce 作业执行单个计算来构建的。如图 2-8 所示，通过创建流经中间 MapReduce 作业系统的数据流，可以创建一个分析步骤的流水线，引导我们得到最终结果。分析人员和开发人员的工作是设计实现 `map` 和 `reduce` 的算法，以得到单一的分析结论，这部分将在第 3 章进行详细探讨。

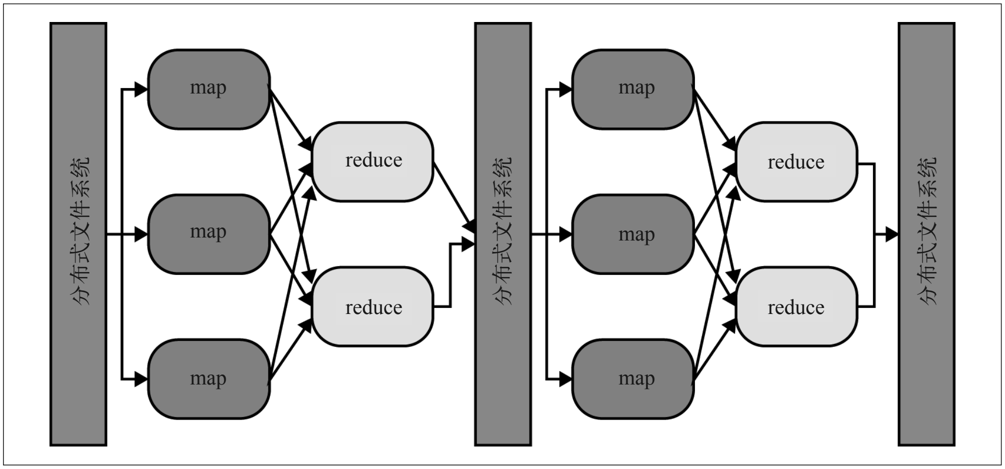


图 2-8：复杂的算法或应用程序实际上是通过 MapReduce 作业链接而成的，其中下游 MapReduce 作业的输入是最近的上游作业的输出

本书探讨如何将计算框架从传统的迭代分析转变为可用于大规模计算的“数据流”。数据流是作业或运算的有向无环图，被用于针对大型数据集实现某种最终计算。最终，大数据应用程序的主要数据工程工作是过滤和聚合大型数据集，以进行最后一英里计算——数据可以放入内存进行计算。显而易见，链式作业适合这种数据处理模型，其他数据处理系统（如 Storm 和 Spark）也与它有关。

2.5 向YARN提交MapReduce作业

MapReduce 的 API 是用 Java 编写的，因此提交给集群的 MapReduce 作业是编译好的 Java 归档（Java Archive，JAR）文件。Hadoop 将 JAR 文件通过网络传输到运行任务（mapper 或 reducer）的每个节点，并执行 MapReduce 作业的各个任务。



虽然本书探索了几种编写 Hadoop 分析作业的方法，但是我们的应用程序将主要通过 Python 编写，使用 MapReduce Streaming 或 Spark。在某些情况下，本书还将使用 Hive 和 Pig 演示在集群上执行数据分析的其他方法。

单词计数示例演示了分布式计算的强大功能，以及如何计算非结构化数据。请从 Hadoop Fundamentals 仓库（https://github.com/bbengfort/hadoop-fundamentals）下载单词计数的 Java 示例程序 WordCount.zip，其中包含以下文件。

WordCount.java
执行作业的 MapReduce 驱动类。

WordMapper.java
发射单词的 mapper 类。

SumReducer.java
统计单词的 reducer 类。

使用如下命令将 Hadoop 作业编译成一个 JAR 文件：

```
hostname $ hadoop com.sun.tools.javac.Main WordCount.java
hostname $ jar cf wc.jar WordCount*.class
```

这会在当前工作目录中创建一个 wc.jar 文件。请注意，这假定几个环境变量已被正确配置，包括 JAVA_HOME 和 HADOOP_CLASSPATH。有关环境变量的详细信息，请参见附录 A。

为了将作业提交到集群并计算莎士比亚作品全集的字数，使用 hadoop jar 命令。该命令连接到 ResourceManager，并发送 wc.jar 文件，使其在集群的所有节点上执行。该命令需要作业归档文件的路径，以及要调用的 main 方法所在的类名。然后，将其他命令行参数传递到作业本身。这个简单程序需要待分析数据的输入路径，以及写入结果的输出路径。输入路径和输出路径都是 HDFS 路径，并且输出路径不能在分布式文件系统上，否则会出现错误（为了防止覆盖或删除集群上的数据）。作业提交如下所示：

```
hostname $ hadoop jar wc.jar WordCount shakespeare.txt wordcounts
```

作业将执行并输出 mapper 和 reducer 的状态，并且在完成时报告作业完成情况的统计信息。一旦完成，作业的结果将写入 wordcounts 目录，可以按如下方式查看该目录：

```
hostname $ hadoop fs -ls wordcounts
```

可以看到几个名字类似于 part-00000 的输出文件。事实上，在计算中使用的每个 reducer 都应该有一个 part 文件。此外，还应该有一个 _SUCCESS 文件和一个 _logs 目录，用于存储有关作业的信息。为了读取作业的结果，针对远程文件系统中的 part 文件执行 cat 命令，并通过管道传输给 less：

```
hostname $ hadoop fs -cat wordcounts/part-00000 | less
```

如果 MapReduce 作业出现问题，你得能停止它。（试想一下，你不小心向 mapper 或 reducer 添加了无限循环或内存密集型进程！）但是键入 Ctrl + C（在 Unix 上发出键盘中断）只能终止显示进度的进程，而不能真正停止作业！ hadoop job 命令让你能管理当前运行在集群上的作业。使用 -list 命令列出所有正在运行的作业：

```
hostname $ hadoop job -list
```

通过输出来标识要终止的作业的 ID，然后通过 -kill 命令终止该作业：

```
hostname $ hadoop job -kill $JOBID
```

与 NameNode 的 Web 接口类似，ResourceManager 也提供了一个 Web 接口来查看作业的状态及日志文件。可以通过托管 ResourceManager 服务的机器的 8088 端口访问

ResourceManager 的 Web UI，此 Web UI 显示所有当前正在运行的作业，以及集群中 NodeManager 的状态。ResourceManager 不跟踪作业的历史记录，但是可以通过 JobHistory 服务器访问历史记录——在托管 JobHistory 服务器的机器的 19888 端口上访问 JobHistory 服务器。

2.6 小结

本章介绍了关于 Hadoop 集群架构的大量细节，并简要介绍了大规模分布式计算系统的需求和实现的许多要点。然而，这并不意味着我们已经介绍了全部内容——相反，目前只是为介绍本书中的概念提供了足够的背景知识。之所以要如此介绍 MapReduce 的概念细节，是为了给开发分布式算法打下基础。在此基础上，本书将继续讨论更复杂的分析算法，让你了解它们的工作原理。不过，本书不会对具体的实现进行更深入的讨论。

因为本书的目标是成为 Hadoop 分布式计算的入门指导，所以它不会关注 Hadoop 集群的搭建、配置或维护，而是关注分析人员与 Hadoop 的交互。因此，下一章将通过 Hadoop Streaming，研究如何使用 Python 编写简单的 MapReduce 分布式作业。

第3章

Python框架和Hadoop Streaming

当前版本的 Hadoop MapReduce 是一个软件框架，用于编写在集群上并行处理大量数据的作业，也是 Hadoop 自带的原生分布式处理框架。该框架提供一个 Java API，允许开发人员指定 HDFS 上的输入输出位置、`map` 和 `reduce` 函数以及其他作业参数（比如**作业配置**）。作业被编译并打包成 JAR 文件，**作业客户端**将其提交给 `ResourceManager`，这一步通常通过命令行完成。然后，`ResourceManager` 调度任务，监控任务，并将状态提供给客户端。

通常，MapReduce 应用程序由 3 个 Java 类组成：`Job`、`Mapper` 和 `Reducer`。mapper 和 reducer 处理键值对计算的细节，通过 shuffle 阶段和 sort 阶段连接。作业通过指定要从 HDFS 序列化的数据的 `InputFormat` 和 `OutputFormat` 类，来配置输入数据和输出数据的格式。所有这些类都必须扩展抽象基类或实现 MapReduce 中的接口。毋庸多言，开发 Java MapReduce 应用程序是非常复杂的。

但是，Java 不是 MapReduce 框架的唯一选择。例如，C++ 开发人员可以使用 Hadoop Pipes，它提供了一个能使用 HDFS 和 MapReduce 的 API。但数据科学家最感兴趣的还是 Hadoop Streaming，这是一个用 Java 编写的实用程序，可以将任何可执行程序指定为 mapper 或 reducer。通过 Hadoop Streaming，shell 实用程序、R 或 Python、脚本都可以用于编写 MapReduce 作业，这使数据科学家可以轻松地将 MapReduce 集成到他们的工作流中，特别是在不需要大量软件开发的日常数据管理任务中。

> Hadoop Streaming 看上去似乎不是 Hadoop 生态系统第一梯队的成员。事实上，大多数 Hadoop 用户在直接使用 Hadoop MapReduce 之前，很可能使用过更高级别的工具，例如 Pig 和 Hive。虽然 Streaming 社区的规模很小，但是有很多框架都是基于它构建的，而且有许多云计算 MapReduce 资源（如 Amazon 的 Elastic MapReduce）原生包含 Streaming。敏捷数据科学利用脚本语言和 Hadoop Streaming 的快速开发，可以快速构建数据分析乃至大规模计算作业，包括机器学习任务。

本章将探讨使用 Hadoop Streaming 的细节，并创建一个小框架，从而使用 Python 快速编写 MapReduce 作业。本章将扩展在第 2 章中使用的简单 WordCount 程序，以便使用 Python 的第三方库进行自然语言处理（natural language processing，NLP）；此外，还会编写一个用于识别文本中重要短语（bigram）频率的 MapReduce 作业；最后，将讨论一些高级的 MapReduce 主题，这对于如何理解 Hadoop，以及如何将这些主题应用于 Python 编写的 Streaming 作业中至关重要。

3.1　Hadoop Streaming

Hadoop Streaming 是一个实用程序，被打包为 Hadoop MapReduce 发行版附带的 JAR 文件。Streaming 作业像普通 Hadoop 作业一样，通过作业客户端传递到集群。但除了可以指定输入和输出的 HDFS 路径的参数外，它还可以指定 mapper 和 reducer 的可执行程序。然后，作业作为普通 MapReduce 作业运行，依然由 `ResourceManager` 和 `MRAppMaster` 管理和监控，直到作业完成。

为了执行 MapReduce 作业，Streaming 利用标准 Unix 流进行输入和输出，因此得名 Streaming。mapper 和 reducer 的输入都是从 `stdin` 读取的，Python 进程可以通过 `sys` 模块访问 `stdin`。Hadoop 要求由 Python 编写的 mapper 和 reducer 将它们输出的键值对写到 `stdout` 中。图 3-1 演示了 MapReduce 中的这个过程。虽然使用 Python 的 Hadoop 开发人员不一定能够通过这种技术访问完整的 MapReduce API（partitioner、输入和输出格式等功能必须用 Java 编写），但这已足以实现数据科学家工作流中许多功能强大的常见作业和任务了。

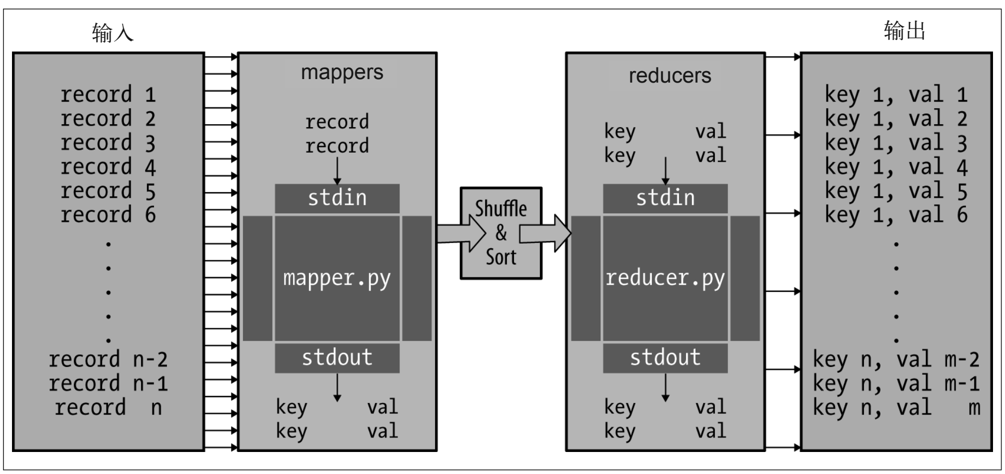


图 3-1：使用 Python 编写的 mapper.py 和 reducer.py 的 Hadoop Streaming 中的数据流

> 不要把 Hadoop Streaming 与 Spark Streaming 或其他使用“无界数据流”的实时计算框架（如 Apache Storm）弄混了。Hadoop Streaming 中的“流”指的是标准的 Unix 流 `stdin`、`stdout` 和 `stderr`，而 Spark Streaming 和 Storm 对一定时间内从窗口流入的数据进行实时分析——它们截然不同！本章所说的“Streaming”具体是指 Hadoop Streaming。

当 Streaming 执行作业时，每个 mapper 任务将在自己的进程内启动提供的可执行文件；然后，将输入数据转换为文本行并将其输送到外部进程的 `stdin` 的同时，从 `stdout` 收集输出。输入数据的转换通常是直接将值序列化，因为数据是从 HDFS 读取的，其中每行都是一个新值。mapper 要求输出是键或值格式的字符串，其中键和值通过某个分隔符分隔，默认为制表符（`\t`）。如果没有分隔符，mapper 就认为输出只有键，值为 `null`。可以通过向 Hadoop Streaming 作业传递参数来定制分隔符。

对 mapper 的输出进行 shuffle 和 sort 之后（确保每个相同的键都发送给同一个 reducer），reducer 也启动了可执行文件。mapper 输出的键值字符串通过 `stdin` 传输到 reducer 作为输入，reducer 的输入和 mapper 的输出相互匹配，并保证按键分组。reducer 发送到 `stdout` 的输出的格式应该与 mapper 的键、分隔符和值的格式相同。

因此，为了使用 Python 编写 Hadoop 作业，需要创建两个 Python 文件：mapper.py 和 reducer.py。只需要在这两个文件中导入 `sys` 模块，就可以访问 `stdin` 和 `stdout`。代码本身需要以字符串的形式处理输入、解析和转换每个数字或复杂的数据类型，我们也需要将输出序列化为字符串。为了演示它是如何工作的，我们将尽可能使用简单的 Python 方法来实现第 2 章讨论的 WordCount 示例。

首先，创建可执行 mapper 文件 mapper.py：

```
#!/usr/bin/env python

import sys

if __name__ == "__main__":
    for line in sys.stdin:
        for word in line.split():
            sys.stdout.write("{}\t1\n".format(word))
```

mapper 只是简单地从 `sys.stdin` 读取每一行，使用空格拆分该行文本，然后逐行将得到的每个单词和数字 1 写入 `sys.stdout`，并用制表符将两者分隔。reducer 则更复杂一些，因为针对每行输入，我们都要记录正在处理哪个键，只有看到一个新键时，才能发射一个完整的和。这是因为与本地 API 不同，单个数据值会在 shuffle 和 sort 期间聚合到流进程中，而不是暴露为一个列表或迭代器。请记住，每个 reducer 任务都可以看到同一个键的所有值，但也可以看到多个键。在 reducer.py 文件中实现的 reducer 如下所示：

```
#!/usr/bin/env python

import sys

if __name__ == '__main__':
    curkey = None
    total = 0
    for line in sys.stdin:
        key, val = line.split("\t")
        val = int(val)

        if key == curkey:
            total += val
        else:
```

```
        if curkey is not None:
            sys.stdout.write("{}\t{}\n".format(curkey, total))

    curkey = key
    total = val
```

当 reducer 迭代 `stdin` 输入中的每一行时，它会根据分隔符拆分该行并将值转换为整数。然后，它执行检查以确保仍在为同一个键计数；如果不是同一个键，则将输出写入 `stdout` 并重新启动新键的计数。mapper 和 reducer 都在“ifmain”块中执行，本章后面会讨论这部分内容。

> 如果你学习过如何使用 Java 进行 MapReduce 编程，你可能会以为 `stdin` 和 MapReduce 的 API 一样，一次只接收一行记录。但是通过 Streaming，mapper 可以访问块中的每一行，并将整个数据集视为单个项目。此外，reducer 也不像在 Java API 中那样接收累积值，而是接收从 mapper 输出的、经过排序的逐行输入。我们将使用 `groupby` 来模拟积累过程，但它不是原生的。

每个 Python 模块都在自己的进程内执行，因此它拥有运行时所有可用的处理和内存资源。但需要注意的是，由于 Hadoop Streaming 把每个 mapper 和 reducer 都视为**可执行**程序，所以每个 Python 文件都应以 `#!/usr/bin/env  python` 开头，从而提示 shell 应该使用 Python，而不是 bash 来解释代码。

既然我们已经充分了解了 Hadoop Streaming 的工作原理，那么就来挑战一下更复杂的代码，想一想可被不同 Streaming 作业重用的高质量 Python 代码，并具体研究如何使用 Hadoop Streaming 来解析 CSV 数据。

3.1.1　使用Streaming在CSV数据上运行计算

虽然 Python 脚本在 Hadoop Streaming 中要做的全部工作只是读取 `stdin` 的内容并将其写入到 `stdout`，但是我们仍然可以改进一下前面的代码，比如可以使用 Python 标准库中的模块进行快速迭代、字符串处理等。本节将开始搭建一个小型的可重用框架，从而为大数据处理需求快速部署 Hadoop 作业。在开始之前，先来看一个读取 CSV 数据的特定示例。

因为 mapper 和 reducer 的输入和输出是字符串，所以我们得仔细思考应该在系统中使用什么数据类型，以及希望 Python 脚本做多少解析工作。例如，可以使用内置的 `ast.literal_eval` 来解析简单数据类型（例如数字、元组、列表、字典或布尔），或者用结构化的序列化格式（例如 JSON 甚至 XML）来输入和输出复杂数据结构。因为 Streaming 逐行进行序列化，所以 Python Streaming 作业非常适用于处理 CSV 文件和其他纯文本文件，在我们的数据集和其他半结构化数据存储中就经常见到这些格式。稍后将讨论其他类型，如 Avro 或其他可以使用的二进制序列化格式。

在这个例子中，我们将使用美国国内航班准点情况数据集。该数据集由美国交通部交通统计局提供，可以从其网站（http://bit.ly/rita-transtats）下载。（本书的 GitHub 仓库中也有一个该数据集整理后的版本。）美国交通统计局提供了一个 CSV 文件，其中包含每个美国国

内航班及其相关运输统计数据，如到达或离港延误，可用于分析。在本例中，整理过后的数据集每行包含的 CSV 数据有：航班日期、航空公司 ID、航班号、始发机场和到达机场、起飞时间和延误分钟、到达时间和延误分钟，最后是空中飞行时间以及里程。

```
2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00
2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00
```

通过计算每个机场的平均起飞延误时间，我们将示范怎样编写结构化的 MapReduce Python 代码。先来看看 mapper，在 mapper.py 文件中写入如下代码：

```
#!/usr/bin/env python

import sys
import csv

SEP = "\t"


class Mapper(object):

    def __init__(self, stream, sep=SEP):
        self.stream = stream
        self.sep    = sep

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def map(self):
        for row in self:
            self.emit(row[3], row[6])

    def __iter__(self):
        reader = csv.reader(self.stream)
        for row in reader:
            yield row


if __name__ == '__main__':
    mapper = Mapper(sys.stdin)
    mapper.map()
```

来逐行看一下这段代码。第一行的 `#!`（声明 shebang）告诉 Linux（具体来说是 bash）使用什么程序来执行这个脚本——本例使用默认环境中的 Python，不管它是什么版本。这么简单的一行代码就创建了可执行脚本，并让 Hadoop Streaming 知道如何处理我们的文件。

后面的几行代码导入了 Python 标准库中的两个模块——`sys`（用于访问 `stdin` 和 `stdout`）以及 `csv`（用于快速解析 CSV 数据）。请注意，因为这些模块都在标准库中，所以在集群中的每个节点上都可以使用它们。第三方包和自定义代码则必须进行特殊处理，之后的内容将讨论这个问题。

我们没有创建一个过程式的脚本处理输入，而是将所有代码都写在 `Mapper` 类中。虽然 Python 实现了函数式编程技术，但它也是一种完全面向对象（object-oriented，OO）的编

程语言。因为 Python 是一种解释型语言，所以它的用途广泛，从用于系统管理的便捷脚本到使用 OO 设计的大规模软件库和代码库，都能创建松耦合系统。在示例代码中，我们使用类创建了一个可扩展的 API，可以将其用于我们所有的 MapReduce 任务。这段代码的目标是可重用和可用于生产。在 3.2 节中，我们将把在此示例中学到的内容结合起来，构建一个完整的微框架，用于部署使用 Python 的 Hadoop Streaming 代码。

航班平均延误时间示例的 `Mapper` 类的实例化需要 `infile` 和 `separator` 这两个参数，它们都有默认值。`infile` 指向接收数据的位置，默认情况下为 `stdin`，这是 Hadoop 的预期值。但你可以将这段代码修改成一个能够分析独立文件的通用框架，让它变得 DRY（don't repeat yourself，不做重复的事情），从而进行各种规模的分析。Hadoop 使用键值对进行计算，因此第二个参数用于确定输入 / 输出字符串的哪个部分是键，哪个部分是值。默认情况下，分隔符是制表符（`\t`）——它是一个模块级“常量”，允许我们在需要时快速重新定义分隔符。

下一个要注意的是在 `mapper` 类上使用的 `__iter__` 内置方法。双下划线通常表示这是 Python 中的一个特殊方法或函数。具体来说，`__iter__` 函数的实现使该类成为可迭代的，该函数返回一个生成器（一般通过 `yield` 语句构造），这是另一个可迭代的对象；该函数如果简单地返回 `self`，就必须同时实现 `next` 或 `__next__` 方法，这两个方法在迭代完成时抛出 `StopIteration`。这个类现在可以用于 `for` 语句，如：

```
for item in Mapper():
    print item
```

执行这行代码时，Python 会调用 `__iter__` 方法来确定如何迭代 `mapper` 类的实例——本例通过使用 `csv.reader` 解析 `stdin` 的每一行，并产生每一行。我们的类在 `map` 方法中将自己作为迭代器，因此可以循环遍历 `self` 中的每一行，也就简单地循环遍历了 `infile` 中的每一行。然后，它将输出始发机场（位置 3）并将此作为键，将起飞延误时间（位置 6）作为值，再使用 `emit` 方法发射出去——简单地将由 `sep` 分隔的键和值作为单行写入 `stdout`。

这段代码的最后一部分是 `if  __name__  =="__main__"` 块，也被称为“ifmain”。在 Python 中，只有脚本作为程序的主入口点运行时，此条件才会被触发，被导入的脚本不会运行该方法。Python 开发人员使用它来判断代码是否在一个库中，或者确保任何执行的代码都在 Python 脚本的底层运行，以方便调试。通过这个语句，可以确保这个块只有在作为 mapper 直接传递给 Hadoop Streaming 时才被执行；如果是导入代码以继承 mapper（例如我们的微框架），这个块就不会被执行。现在来看看 reducer，在 reducer.py 文件中写入如下代码：

```
#!/usr/bin/env python

import sys

from itertools import groupby
from operator import itemgetter

SEP = "\t"


class Reducer(object):
```

```
    def __init__(self, stream, sep=SEP):
        self.stream = stream
        self.sep    = sep

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def reduce(self):
        for current, group in groupby(self, itemgetter(0)):
            total = 0
            count = 0

            for item in group:
                total += item[1]
                count += 1
                self.emit(current, float(total) / float(count))

    def __iter__(self):
        for line in self.stream:
            try:
                parts = line.split(self.sep)
                yield parts[0], float(parts[1])
            except:
                continue

if __name__ == '__main__':
    reducer = Reducer(sys.stdin)
    reducer.reduce()
```

`Reducer` 类与 `mapper` 类非常相似，但我们在这段代码中引入了一些新的项目——一个内存安全的迭代器帮助函数 `groupby` 和一个操作符函数 `itemgetter`。和 `mapper` 一样，在 `reduce` 函数中也使用一个迭代器遍历整个数据集，分割键和值。在本例中，键和值是使用 `separator` 分割的。键是由分隔符拆分的第一个项目，使用 Python 切片；值是第一个分隔符之后的其他所有内容。这与 Hadoop Streaming 处理 map 任务的输出时的默认行为相匹配，即将第一个制表符前面的所有字符作为键，剩下的都作为值。由于本例使用了浮点除法来计算平均值，所以简单地将字符串值解析为浮点数。

在 Python 代码中考虑错误处理是极其重要的。比如，如果输入数据损坏了（不是浮点数或不可分析），那么将值解析成浮点数时极易发生 `ValueError` 异常。请注意，异常处理在处理大数据集时至关重要。一种常用的策略是跳过引发异常的行，因为还有大量的数据需要计算。

经过 Hadoop 流水线中的 shuffle 阶段和 sort 阶段之后，进入 reducer 的数据的键是按字母顺序排序的，所以我们希望自动将键及其值组合在一起。`groupby` 方法以内存安全的方式实现了这一目标，让你能访问键，并像访问列表一样访问值。内存之所以是安全的，是因为 `groupby` 返回的不是一个保存在内存中的列表，而是一个迭代器，并且一次只读取一行（因此确保大数据集不会让 worker 节点的资源容量不堪重负）。`itemgetter` 函数简单地指定了应该根据每个元组的哪个值进行分组——在本例中，是元组的第一个元素。

在将值进行高效的内存分组之后，简单地将延误时间相加，除以航班数，然后发射机场作为键、平均值作为值的输出。虽然代码变得冗长了，但希望创建微框架的过程能更加清晰，微框架能消除这段代码中的大部分重复代码，并使其可重用。

3.1.2 执行Streaming作业

在介绍如何通过将作业提交到作业客户端从而在 Hadoop 集群上执行 Streaming 作业之前，先来看一个不产生 Hadoop 集群开销的脚本测试高招。因为 Streaming 使用 Unix 标准管道，所以可以使用 Linux 管道和 `sort` 命令模拟 Hadoop MapReduce 流水线。

要测试代码，请先确保 mapper.py 和 reducer.py 是可执行的。只需在终端中使用 `chmod` 命令，如下所示：

```
hostname $ chmod +x mapper.py
hostname $ chmod +x reducer.py
```

要通过 CSV 文件作为输入来测试 mapper 和 reducer 的话，可以使用 `cat` 命令输出文件的内容，通过管道将输出从 `stdout` 传输到 mapper.py 的 `stdin`，再传输到 `sort`，然后到 reducer.py，最后将结果打印到屏幕上。要测试上一节中计算每个机场平均延误时间的 mapper 和 reducer 的话，可以在终端中执行以下命令，其中 mapper.py、reducer.py 和 flights.csv 都在当前工作目录中：

```
hostname $ cat flights.csv | ./mapper.py | sort | ./reducer.py
ABE     -3.57142857143
ABI     55.375
ABQ     3.83333333333
ABR     -4.0
ABY     -1.33333333333
ACT     -8.2
ACV     109.142857143
ACY     -8.0
ADQ     -14.0
AEX     -6.55555555556
AGS     31.4
ALB     -1.5
ALO     -8.5
AMA     0.8
...
TWF     -7.0
TXK     -4.66666666667
TYR     -6.71428571429
TYS     12.9583333333
VEL     -7.5
VLD     -5.0
VPS     5.06666666667
WRG     -3.75
XNA     14.2580645161
YAK     -17.5
YUM     -0.222222222222
```

Unix 管道是一种测试 Hadoop Streaming 的 mapper 和 reducer 的方法，既简单又有效，还

能有效说明集群是如何使用 mapper 和 reducer 代码的。这种方法非常适合在编写脚本时进行快速测试，因为你不用等待 Hadoop Streaming 作业完成，也不需要解析 Java 调用过程（traceback）。如果你在进行测试驱动开发（这是敏捷数据科学的自然补充），则可以使用 `Popen` 模拟管道进行集成测试。

> 在下面的示例中，我们使用 `$HADOOP_HOME` 之类的环境变量指定特定的路径或配置。尽管这些环境变量的名称在每个 Hadoop 发行版中可能有所不同，但它们通常在发行版安装时就已经被设置好了。本书示例假设你使用的是伪分布式的节点设置，如附录 A 所述。

为了将代码部署到集群，需要将 Hadoop Streaming JAR 提交给作业客户端，并传入自定义的操作符参数。Hadoop Streaming 作业的位置取决于 Hadoop 集群的设置。现在假设你设置了环境变量 `$HADOOP_HOME` 并且 `$HADOOP_HOME/bin` 在 `$PATH` 中，`$HADOOP_HOME` 指定了 Hadoop 的安装位置。这样就可以按照如下所示的方法在集群上执行 Streaming 作业：

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input flights.csv \
    -output average_delay \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py
```

请注意，这里使用了 `-file` 选项，它让 Streaming 作业往集群上发送脚本（否则程序无法在节点上找到这些脚本）。执行此命令将在 Hadoop 集群上启动该作业。mapper.py 脚本和 reducer.py 脚本将在处理之前被发送到集群中的每个节点，并应用于流水线的每个阶段。

如果有需要与作业一起发送的其他文件（如航空公司 ID 的查找表），可以使用 `-file` 选项将它们与作业打包在一起。代码中使用的任何第三方依赖也应与作业一起被提交，通常打包在 Python ZIP 文件中。对于较大的依赖文件（例如 NLTK）或者需要使用 Cython 编译的依赖（例如 NumPy 或 SciPy），则需要在作业启动之前在每个节点的系统路径中安装相应依赖。

Hadoop Streaming 有许多其他设置，允许用户指定 Hadoop 库中的类作为 partitioner、输入和输出格式等。但是 Hadoop Streaming 也可以使用 Python 脚本作为 combiner，这在大数据分析中尤为重要。只需使用 `-combiner` 选项即可指定 combiner。一种方法是将 mapper 更新为流水线，使用相同的 shell 脚本、`sort` 和 reducer，和在本地测试脚本一样。不过，指定另一个 Python 脚本作为 combiner 通常更有效，因为大多数 combiner 都与 reducer 完全相同或非常相似。

3.2 Python的MapReduce框架

Hadoop Streaming 稍微高级一点的用法是利用标准错误流（`stderr`）更新 Hadoop 状态以及 Hadoop 计数器。这种技术本质上是让 Streaming 作业访问 `Reporter` 对象——MapReduce Java API 的一部分，用于跟踪作业的全局状态。通过将特殊格式的字符串写入 `stderr`，

mapper 和 reducer 可以更新全局作业状态，以报告进度并表明它们是活动的。对于需要大量时间的作业（尤其是涉及从作业的 pickle 文件中加载大型模型的任务），确保框架不会认为任务已超时至关重要。

计数器在整个 MapReduce 框架或应用程序范围内进行全局聚合，以键值对的形式保存数值。这在许多任务中都非常有用，能使分析人员和开发人员了解系统在数据分析期间发生了什么。计数器可以通过满足结合律的运算来累加，这本质上就增加了计数器的值。虽然 Hadoop 实现了多个计数器，能对处理的记录和字节数进行计数，但是自定义计数器能更轻松地跟踪作业中的指标数据或提供副计算的相关渠道。

例如，我们可以在简单的 WordCount 程序中实现计数器，以记录全局单词数以及我们的**词汇量**（即不同单词的个数），从而进行最终计算——**词汇多样性**。词汇多样性是单词个数与词汇量的比率，表示单个单词在文本中的平均出现频率。这类指标数据对于理解语料库的变化对自然语言处理应用程序的影响至关重要。副计算指标数据，也就是这里的计数器，追踪 Hadoop 作业的主体，可以用作输出，但不影响主要的计算。当评估跨数据集的机器学习模型时，也可以用它们来计算均方误差或分类指标。

要使用 `Reporter` 的 `Counter` 和 `Status` 功能的话，可以为上一节中的 `Mapper` 和 `Reducer` 类添加如下方法：

```
def status(self, message):
    sys.stderr.write("reporter:status:{}\n".format(message))

def counter(self, counter, amount=1, group="ApplicationCounter"):
    sys.stderr.write(
    "reporter:counter:{},{},{}\n".format(group, counter, amount)
    )
```

`counter` 方法允许 `map` 和 `reduce` 函数更新任意命名计数器的计数。根据需要，更新的值可为任意值（默认为递增 1）。可以将计数器的组设置为任意名称，通常默认为应用程序的名称。与之类似，`status` 方法允许 MapReduce 应用程序向框架发送任意消息，并使它们在日志或 Web 用户界面中可见。

为了扩展航班平均延误应用程序，让它提供准点和延迟航班的计数，并在开始和结束时发送状态更新，按如下所示更新 `map` 函数：

```
def map(self):
    self.status("mapping started")
    def map(self):
        for row in self:
            if row[6] < 0:
                self.counter("early departure")
            else:
                self.counter("late departure")

            self.emit(row[3], row[6])

    self.status("mapping complete")
```

仅仅通过添加这几行，我们就能更好地了解平均延误程序是如何运行的，而不需要专门编写

冗长的 Hadoop 作业计算准点和晚点航班的计数。我们可能会想在 reducer 中计算有多少机场有航班数据。因为 reducer 将看到数据集中的每个机场，因此可以这样更新 `reduce` 函数：

```
def reduce(self):
    for current, group in groupby(self, itemgetter(0)):
        self.status("reducing airport {}".format(current))
        ...

        self.counter("airports")
        self.emit(current, float(total) / float(count))
```

随着分析应用程序规模的增长，这些能实现 Hadoop Streaming 全部功能的技术将变得至关重要。再以自然语言处理为例，为了进行词性标注或命名实体识别，应用程序必须将 pickle 之后的模型加载到内存中。此过程可能持续几秒钟到几分钟——使用状态机制警告框架任务仍在正常运行，可以确保集群不会因为推测执行机制而瘫痪。即使在运行其他作业时，计数器也能帮助应用程序从全局范围分析大型数据集。

既然提到了全局范围，就来说说最后一个能改进由 Python 编写的 Streaming 应用程序的工具：作业配置变量（job configuration variable，简称“JobConf 变量”）。Hadoop Streaming 应用程序自动将作业的配置变量添加到环境中，用下划线（`_`）替换圆点（`.`）来重命名配置变量。例如，如果要访问作业中 mapper 的数量，可以请求 `"mapred.map.tasks"` 配置变量。虽然这个示例不一定有用，但用户定义的配置值可以使用圆点形式的 `-D` 参数提交到 Hadoop Streaming 中，其中可包含重要信息，如共享资源的 URL。要在 Python 代码中访问作业配置变量，可添加以下函数：

```
import os

def get_job_conf(name):
    name = name.replace(".", "_").upper()
    return os.environ.get(name)
```

很明显，使用一个微型、可重用的框架对 Hadoop Streaming 的 Python 开发大有帮助。该框架应该有一个用于处理 mapper 和 reducer 的 Streaming 细节的基类，以及应该在自定义 MapReduce Streaming 作业中扩展的抽象基类 `Mapper` 和 `Reducer`。想想下面的框架：

```
import os
import sys

from itertools import groupby
from operator import itemgetter

SEPARATOR = "\t"


class Streaming(object):

    @staticmethod
    def get_job_conf(name):
        name = name.replace(".", "_").upper()
        return os.environ.get(name)
```

```
    def __init__(self, infile=sys.stdin, separator=SEPARATOR):
        self.infile = infile
        self.sep = separator

    def status(self, message):
        sys.stderr.write("reporter:status:{}\n".format(message))

    def counter(self, counter, amount=1, group="Python Streaming"):
        msg = "reporter:counter:{},{},{}\n".format(group, counter, amount)
        sys.stderr.write(msg)

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def read(self):
        for line in self.infile:
            yield line.rstrip()

    def __iter__(self):
        for line in self.read():
            yield line


class Mapper(Streaming):

    def map(self):
        raise NotImplementedError("Mappers must implement a map method")


class Reducer(Streaming):

    def reduce(self):
        raise NotImplementedError("Reducers must implement a reduce method")

    def __iter__(self):
        generator = (line.split(self.sep, 1) for line in self.read())
        for item in groupby(generator, itemgetter(0)):
            yield item
```

在编写传递给 Hadoop Streaming 的 mapper 和 reducer 时，只需在 Streaming 作业中引入这个文件，并从该框架中导入合适的类。在扩展类后，只需在代码中实现 `map` 函数或 `reduce` 函数即可。下一节描述了一个具体的示例——将此框架与自然语言工具包（natural language toolkit，NLTK）结合使用，以执行更精确的单词计数。

3.2.1　短语计数

一直以来，Hadoop 程序的“Hello, World”都是单词计数程序。使用 Python 代码对文件执行单词计数是演示分布式计算的好方法，但是有了 Hadoop Streaming 的话，就可以简单地使用 Linux 的 `wc` 命令，因为这条命令也从 `stdin` 接收输入，并输出到 `stdout`。通过使用 mapper 和 reducer，Python 允许我们使用多个 reducer 任务进行数据聚合，将处理极大数据集的工作分解开来。此外，单词计数是语言处理所用的统计方法的基础，我们可

以使用 Python 中可用的高级文本处理技术来进行更高级的词法分析，例如使用词形还原（lemmatization）技术进行短语计数或更高级的创建索引的方法。

Hadoop Streaming 非常适用于文本处理，不仅因为通过它能访问 TextBlob 和 NLTK 等库，还因为 Hadoop Streaming 本身就以逐行方式使用字符串序列。默认情况下，Hadoop Streaming 期望通过 Streaming 作业的标准输入和标准输出的文本是由制表符分隔的——如果你还能将数据视为键值对，那当然更好，不过也不是必须的。

> NLTK 和 TextBlob 是第三方依赖，这意味着 Python 默认不包含它们。要安装这些包，可以使用 Python 包管理器 `pip`，并且集群中的**每个**节点都必须安装这些依赖。为了简化问题，假定所有额外的库都已经安装在集群上，不过集群管理不在本书的范围之内。如果你使用的是伪分布式设置，那么执行 `pip install nltk` 应该就能完成 NLTK 的安装。

使用 Python 微框架来编写一个稍加改进的 MapReduce 应用程序，执行单词计数。首先，将单词全部归范化为小写字母，像“Apple”这样的单词将与“apple”相同。不是所有语言处理应用程序都可以这么做——英语（和许多其他语言）的大小写是语法的重要部分，表明句子开始或专有名称（例如 Apple Paltrow 或 Apple, Inc.）。然而，在词汇评估中采用规范化倒是没什么问题，Apple 在句子开头还是作为专有名词在这里无关紧要。

下一步，从单词计数中去掉标点符号和停用词。停用词是语言中的虚词，例如冠词（“a”或“an”）、限定词（“the”“this”“my”）、代词（“his”“they”）和介词（“over”“on”“for”）。由于停用词具备这种功能性用法，所以它们的出现非常频繁，占据了语料库的一大部分。据说，对一些信息检索应用程序来说，停用词是给定词汇分布中最常见的词，所以被自动去除以提高应用程序的性能。在这种情况下，“want”或“has”这样的普通动词可能会被排除在外。不管是去除停用词、标点符号，还是文本的标准化，都可以大幅降低词汇密度，对了解特定语料库中最重要的单词大有帮助。

最后，使用这个归范化的语料库来统计**短语**的个数，也就是经常一起出现的单词（例如忽略停用词后连续两次一起出现）。统计学家通过短语分析来获取通常一起出现或可能具有某种特殊意义的词语，例如“lawn chair”（草坪躺椅）或“vetoed bill”（否决议案）。短语是 *n*-Gram 语言模型中最简单的形式。*n*-Gram 是一种模型构建技术，可以预测给定上下文的下一个单词。

使用之前的框架，`Mapper` 完成了大部分工作：

```
#!/usr/bin/env python

import sys
import nltk
import string

from framework import Mapper

class BigramMapper(Mapper):
```

```
    def __init__(self, infile=sys.stdin, separator='\t'):
        super(BigramMapper, self).__init__(infile, separator)

        self.stopwords = nltk.corpus.stopwords.words("english")
        self.punctuation = string.punctuation

    def exclude(self, token):
        return token in self.punctuation or token in self.stopwords

    def normalize(self, token):
        return token.lower()

    def tokenize(self, value):
        for token in nltk.wordpunct_tokenize(value):
            token = self.normalize(token)
            if not self.exclude(token):
                yield token

    def map(self):
        for value in self:
            for bigram in nltk.bigrams(self.tokenize(value)):
                self.counter("words") # 计算短语的总数
                self.emit(bigram, 1)


if __name__ == "__main__":
    mapper = BigramMapper()
    mapper.map()
```

map 方法很直观。它循环遍历整个输入，并使用 nltk.wordpunct_tokenizer 对值进行分词（tokenize），值是数据集的一行文本。这个分词器确保不仅单词（包括缩写）会被拆分，标点符号也会被拆分。通过使用 nltk 内置的停用词语料库，分词器还能忽略标点符号和停用词。请注意，使用自定义停用词列表改进此代码并不难，可以使用 -file 参数将该列表与作业一起打包。

为了执行短语计数，需要发射键为令牌、值为 1 的元组。我们创建了一个简单的 emit 帮助函数，它将由分隔符字符串分隔的键值对写入到输出（在本例中是 stdout）。可以使用内置的 nltk.bigrams 函数收集短语。

reducer 实现了一个非常常见的 MapReduce 模式——SumReducer。这类 reducer 的使用频率非常高，你甚至都想把它当作一个标准类，与 IdentityMapper 和其他标准模式（你会在第 5 章见到这些模式）一道添加到微框架中。SumReducer 的代码如下所示：

```
#!/usr/bin/env python

from framework import Reducer


class SumReducer(Reducer):

    def reduce(self):
        for key, values in self:
            total = sum(int(count) for count in values)
```

```
            self.emit(key, total)


    if __name__ == '__main__':
        reducer = SumReducer()
        reducer.reduce()
```

请注意，这个 reducer 忽略了键——文本字符串形式的短语元组，因为 reducer 只计算该元组的出现次数。然而，如果为了改变键空间（例如基于第一个单词过滤短语）而需要处理这个复合键，或者因为复合键在链式 MapReduce 作业的一个 mapper 中而需要处理这个复合键，可以使用 Python 的 `literal_eval` 将字符串转换为元组：

```
import ast

key = ast.literal_eval(key)
```

使用与之前示例相同的命令，通过 Hadoop Streaming 将此作业提交到集群，但要确保 framework.py 文件也被打包并随作业一起发送。作业提交命令如下所示：

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input corpus \
    -output bigrams \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py \
    -file framework.py
```

请注意，在这个例子中，假设第三方依赖 `nltk` 已被经安装在集群中的每个节点上——如果你有集群的管理访问权限，或者能从特定的 AMI 启动集群，就可以做到。否则，`nltk` 需要打包成 ZIP 文件，并使用 `-file` 参数发送到集群。

3.2.2　其他框架

虽然我们已经创建了一个可以编写 MapReduce 作业的小框架，但还有其他一些框架也能让你使用 Python 编写 MapReduce 作业，这点十分重要。编写本书时，最流行的两个框架是 Yelp 的 mrjob（http://bit.ly/1NqmsvA）和 GitHub 上的 dumbo（http://bit.ly/1UQx8G3），它们封装了 Hadoop Streaming 并添加了更多的功能。其他框架还有封装了 Hadoop Pipes（Hadoop 的 C++ API）的 pydoop（http://bit.ly/1LizcVD）和使用 Cython 封装了 Streaming 的 hadoopy（http://bit.ly/1TQugXl）。

这些框架试图帮助 Python 开发人员编写 Hadoop 作业，但是却造成了性能损失。它们提供了更简单的 API、编程接口以及 Python 中的标准工具，甚至还提供了运行和启动作业的方法，让开发人员能更专注于 Python 开发而不是 Hadoop 集成。更高级的框架有 `TypedBytes`，这是一种 Hadoop 中的二进制序列化格式，支持将 Python 对象序列化为输入和输出，能显著提高这些框架的性能。

`mrjob` 库也值得关注一下，因为 Yelp 正在积极开发它，其平台完全在 Amazon Web Services 生态系统内。因此，`mrjob` 是唯一适合通过 Python 的 `boto` 库在 Amazon 的 Elastic MapReduce

框架上快速部署和运行作业的库。使用 `mrjob` 编写的作业通常是包含完整 MapReduce 代码的单个文件，可以直接在本地文件系统、EMR 或常规的 Hadoop 集群上执行。此外，可以通过简单的配置文件配置作业。

`dumbo` 库是最早的 Python Hadoop Streaming 框架之一，虽然没有被经常维护，但使用广泛。Tom White 编写的《Hadoop 权威指南》认为它是可选的框架。`dumbo` 框架完全封装了 Hadoop Streaming，并使用 `TypedBytes` 来提高性能。通过它，可以高效地编写复杂的链式 MapReduce 作业。它还附带用于管理和执行作业、提供与 HDFS 交互的命令行脚本。

最后，简单地使用 Hadoop Streaming 是迄今为止性能最优的解决方案，因为它不依赖于第三方库，而且足够轻巧，可以部署在各种分析场景中。本书中的 MapReduce 示例将使用本章描述的 Streaming 机制。对于更大、更复杂的分析，开发人员需要评估这些框架，使其成为工作流程的一部分。

3.3 MapReduce进阶

这一节将介绍一些与 MapReduce 紧密相关的高级主题，引入一些在 MapReduce 算法和优化中发挥重要作用的概念，因为你在阅读其他有关如何实现不同分析的资料时会遇到这些术语。这里不会介绍如何使用这些工具，而是从**概念**层面去介绍，以便在你对 MapReduce 进行深入探索时，不会对它们感到陌生。

这些工具很难在没有 Java API 的情况下实现，因此不适合将它们放入介绍 Hadoop Streaming 的章节中，但是在讨论 MapReduce 时对它们避而不谈又实在是说过不去。我们将在后文讨论 combiner（主要的 MapReduce 优化技术）、partitioner（确保在 reduce 步骤中不出现瓶颈的技术）和作业链（用于组合更大的算法和数据流的技术）。

3.3.1 combiner

mapper 会产生大量的中间数据，这些中间数据必须通过网络传输，进行 shuffle、sort 和 reduce。由于网络是物理资源，大量数据的传输可能会导致作业延迟和内存瓶颈（比如 reducer 要保存到内存中的数据太多）。combiner 是解决这个问题的主要机制，而且它本质上也是与 mapper 输出相关联的中间 reducer。在将数据转发到合适的 reducer 之前，combiner 通过执行一个 mapper 局部的 reduce 来减少网络流量。例如，两个 mapper 和一个简单的求和 reducer 产生如下输出。

mapper 1 的输出：

```
(IAD, 14.4), (SFO, 3.9), (JFK, 3.9), (IAD, 12.2), (JFK, 5.8)
```

mapper 2 的输出：

```
(SFO, 4.7), (IAD, 2.3), (SFO, 4.4), (IAD, 1.2)
```

求和 reducer 的目标输出：

```
(IAD, 29.1), (JFK, 9.7), (SFO, 13.0)
```

每个 mapper 都为 reducer 带来额外的工作，即每个 mapper 都会产生重复的键。combiner

预先计算每个键的和，减少生成的键值对的数量，从而减少网络流量。此外，因为存在较少的重复键，所以 shuffle 操作和 sort 操作也变得更快。

只要运算满足交换律和结合律，combiner 和 reducer 就是相同的——这很常见，但也不总是这样。只要 combiner 的输入输出数据类型和 mapper 的输出数据类型一样，则 combiner 可以执行任意的局部聚合（partial reduction）。因此，combiner 的运算与 reducer 不同时，算法常常同时使用 mapper、reducer 和 combiner 实现。要在 Hadoop Streaming 中指定 combiner，可以使用 `-combiner` 选项，与指定 mapper 和 reducer 类似：

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input input_data \
    -output output_data \
    -mapper mapper.py \
    -combiner combiner.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py \
    -file combiner.py
```

如果 combiner 与 reducer 匹配，则只需将 reducer.py 文件指定为 combiner 即可，无须添加额外的 combiner 文件。在我们创建的微框架中，`combiner` 类会简单地继承 `Reducer`。

3.3.2 partitioner

partitioner 通过划分键空间来控制如何将键及其值发送到每个 reducer，默认使用的 `HashPartitioner` 通常就能满足需求。它通过计算键的散列值并将键分配给由 reducer 数量确定的键空间，来将键均匀地分配给每个 reducer。给定均匀分布的键空间后，每个 reducer 将获得相对平均的工作负载。

一旦键的分布不平均，比如大量的值与一个键相关联，其他键几乎没有关联的值，问题就出现了。在这种情况下，大部分 reducer 的工作量不饱满，并行 reduce 的大多数好处也就无从体现。一个自定义的 partitioner 可以根据散列之外的其他语义结构（通常是特定于领域的）划分键空间，从而缓解这个问题。某些类型的 MapReduce 算法也可能需要自定义 partitioner，最典型的就是实现左外连接。最后，因为每个 reducer 都将输出写入自己的 part-* 文件，使用自定义 partitioner 还能支持更清晰的数据组织，让你根据分区条件将分段输出写入每个文件，例如写入按年输出数据。

不幸的是，只能使用 Java API 创建自定义 partitioner。不过 Hadoop Streaming 用户仍然可以从 Hadoop 库中指定 partitioner Java 类，或者编写自己的 Java partitioner 并将其与 Streaming 作业一起提交。

3.3.3 作业链

大多数复杂的算法不能使用简单的 `map` 和 `reduce` 描述。因此，为了实现更复杂的分析，需要一种被称为**作业链**的技术。如果可以将复杂的算法分解成几个较小的 MapReduce 任务，那么将这些任务链接在一起就可以产生完整的输出。考虑计算数据集中变量对的 Pearson 相关系数。要得到 Pearson 相关系数，需要计算每个变量的平均值和标准差。采用单个

MapReduce 无法轻松实现这个目标，所以可以采用以下策略。

(1) 计算每一个 (*X*, *Y*) 的平均值和标准差。
(2) 使用第一个作业的输出来计算协方差和 Pearson 相关系数。

平均值和标准差可以在初始作业中计算：在 mapper 中计算总个数、和以及平方和，然后在 reducer 中计算平均值和标准差。第二个作业取第一个作业输出的平均值和标准差，在 mapper 中将各个值和平均值的差值相乘来计算协方差，然后通过适当求和和取平方根来计算出该相关系数。如图 3-2 所示，第二个作业依赖于第一个作业。

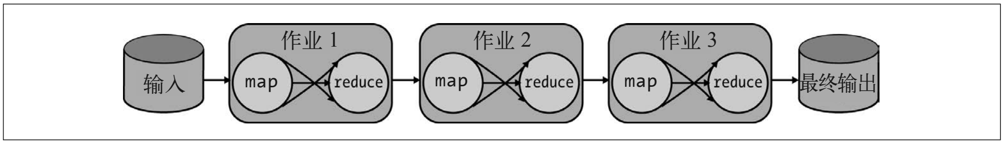


图 3-2：线性作业链将一个或多个 MapReduce 作业的输出作为输入发送到另一个作业来产生完整的计算

因此，作业链是许多小作业的组合，通过将一个或多个作业的输出发送给另一个作业作为输入，从而实现完整的计算。为了实现这样的算法，开发人员必须考虑每一步计算怎样 reduce 出中间值——不仅仅是 mapper 和 reducer 之间的中间值，还包括作业之间的中间值。如图 3-2 所示，许多作业都被认为是**线性作业链**。线性依赖性意味着每个 MapReduce 作业仅依赖于前一个作业。然而，这是一种简化的作业链形式，更普遍的作业链表示为作业依赖于一个或多个先前作业的**数据流**。可以用有向无环图（directed acyclic graph，DAG）来表述复杂作业，它描述数据如何从输入源通过每个作业（有向部分）流向下一个作业（从不重复步骤，无环部分），最后作为最终输出（如图 3-3 所示）。

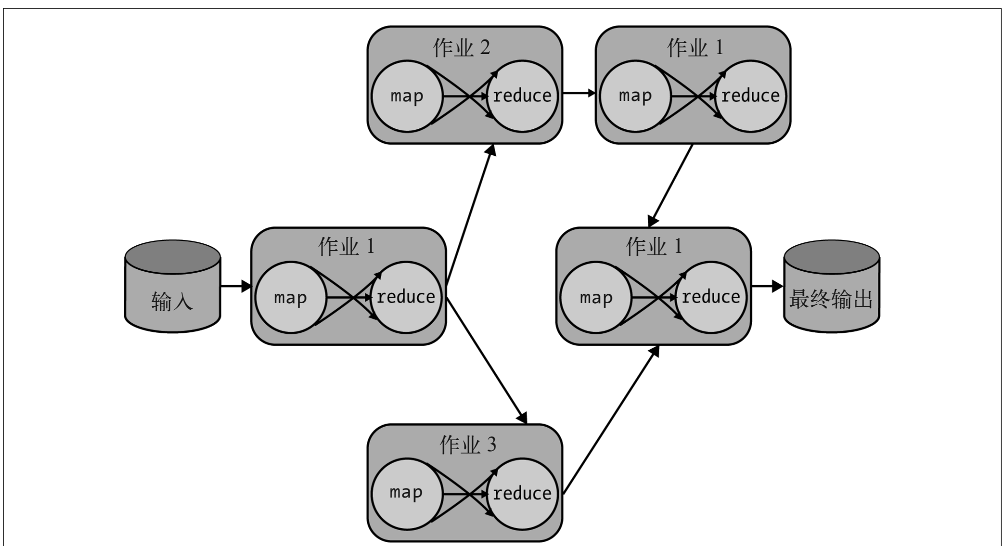


图 3-3：数据流作业链是线性作业链的扩展

仅有 map 的作业

在考虑作业链和作业依赖时，注意可能会有仅有 map 的作业。仅有 map 的作业分两种：不需要聚合的作业，以及积极避免 shuffle 阶段和 sort 阶段的作业——要么为了保持数据的顺序，要么为了优化作业的执行。

要执行仅有 map 的作业，只需将 reducer 的数目设置为 0 即可。有了 Hadoop Streaming，就可以使用 `-numReduceTasks` 标志指定 reducer 的数目。通过使用 identity mapper，也可以实现仅有 reduce 的作业，第 5 章会讨论这个问题。“仅有 map”的作业可以使用 identity reducer 实现排序。

为了计算 Pearson 相关系数并说明作业链，我们将使用由维基百科给出的公式计算样本中的 Pearson 相关系数。输入的数据是键值对，其中键是因变量 y，值是因变量的向量，要求变量 x_i 与 y 的相关性。

等式 3-1 计算样本的 Pearson 相关系数的公式

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\ \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

第一个 MapReduce 作业将计算 n 以及 x 和 y 的平均值，如下所示：

```
class VariablePairsMapper(Mapper):

    def map(self):
        # 计算(x, y)的个数及和
        # 输出键是x在vector中的索引
        for y, vector in self:
            for x, i in enumerate(vector):
                self.emit(i, (1, x, y) )


class PairsMeanReducer(Reducer):

    def reduce(self):
        for key, values in self:
            # 将所有的值加载到内存,这样可以迭代两次
            values = list(values)

            # 计算(x, y)的和及元素的个数
            sx, sy, sn = 0
            for (n, x, y) in values:
                sn += n
                sx += x
                sy += y

            # 计算x和y的平均值
            xbar = sx / n
            ybar = sy / n

            # 发射每一个(x, y)及x和y的平均值
            for (n, x, y) in values:
                self.emit(key, (x, y, xbar, ybar))
```

有了平均值之后，就可以计算协方差和标准差，以便在第二个作业中计算 Pearson 相关系数。为此，需要再次传入**相同的**输入数据，并将第一个作业的输出作为该作业的输入。为了简化以说明问题，将每个 (x, y) 及其相关联的均值作为第一个作业的输出：

```
import math


class PearsonMapper(Mapper):

    def map(self):
        # 计算x和xbar的差及y和ybar的差
        # 发射差的乘积及其平方
        for i, (x, y, xbar, ybar) in self:
            xdiff = x-xbar
            ydiff = y-ybar
            self.emit(i, (xdiff*ydiff, xdiff**2, ydiff**2))


class PearsonReducer(Reducer):

    def reduce(self):
        for key, values in self:
            # 计算差乘积的和以及平方的和
            sxyd = 0
            sxd2 = 0
            syd2 = 0

            for (xyd, x2d, y2d) in values:
                sxyd += xyd
                sxd2 += x2d
                syd2 += y2d

            # 发射相关系数
            r = sxyd / (math.sqrt(sxd2) * math.sqrt(xyd2))
            self.emit(key, r)
```

虽然这不是实现并行计算 Pearson 相关系数的最有效方法，但它展示了一个清晰的作业链。首先要考虑的问题是，如何输出作业 1 的数据，才能让作业 2 运行。第 5 章将从这个简单的引子深入到更复杂的主题，探索 pair 和 stripe，让这样的复杂计算变得更可行。虽然可以人工设置作业链（在命令行上先执行第一个作业，然后执行第二个作业），但这不是最理想的办法。在第 8 章中，我们将探讨**数据流**，以及如何使用高级工具将作业链接起来。

3.4 小结

Hadoop Streaming 是重要的工具，让使用 R 或 Python（而不是 Java）编程的数据科学家能够立即开始使用 Hadoop，特别是 MapReduce。长久以来，如果你想使用 Python 处理大数据，除了 Hadoop Streaming 别无他法。但对于更复杂的作业或算法，特别是那些需要使用 combiner 或 partitioner 进行优化的作业，就需要使用 Java API 了。

然而此时，事情出现了转机，特别是对 Python 开发人员来说。下一章将讨论 Spark——一个与众不同的 Hadoop 计算框架，附带了原生的 Python API（很快就会有 R API）。Spark 正迅速成为数据科学平台的首选，很大一部分是因为 DataFrame 和大量分析包等工具正在 Spark 上构建。

但是，MapReduce 和 Hadoop Streaming 没有完全被 Spark 包含。实际上，如果仅仅就内置的 shuffle 和 sort 而言，其实更适合用 MapReduce 处理批处理作业，特别是那些经常运行的作业（例如 ETL 操作或其他数据整理和清理过程）。此外，MapReduce 和 Hadoop Streaming 经过精心构建与层层测试，可以为任务关键型应用程序所信任。事实上，因为编程模型间太过相似，现在大多数的大数据会同时使用 MapReduce 和 Spark，二者都能很好地满足其特定类型的应用程序。

第 4 章

Spark内存计算

在过去十年中，HDFS 和 MapReduce 一直是大规模机器学习、大规模分析和大数据设备的基石和驱动力。与大多数平台技术一样，Hadoop 的成熟已经带来了一个稳定且通用的计算环境，足以为图形处理、微批处理、SQL 查询、数据仓储和机器学习等任务构建专用工具。然而，Hadoop 越普及，越要求它具备能应对更广泛的新场景的专业化能力。并且我们越发觉得 MapReduce 的批处理模型不太适合常见的工作流，包括针对单个数据集的迭代、交互和按需计算。

主要的 MapReduce 抽象（将计算规定为 map 和 reduce）是并行的，它易于理解，并且隐藏了分布式计算的细节，从而保证 Hadoop 的正确性。然而，为了实现协调性和容错性，MapReduce 模型使用拉执行模型，需要将中间数据写回 HDFS。不幸的是，将数据从其存储位置移动到计算位置所需要的 I/O 在任何计算系统中都是最大的时间成本。因此，MapReduce 在具有极高的安全性和弹性的同时，运行任务的速度也不可避免会慢一些。更糟的是，几乎所有应用程序都必须在多个步骤中将多个 MapReduce 作业链接在一起，从而创建面向最终所需结果的数据流。这导致不为用户所需的大量中间数据被写入 HDFS，从而产生额外的磁盘开销。

为了解决这些问题，Hadoop 采用了更通用的资源管理框架进行计算，这便是 YARN。以前，MapReduce 应用程序分配给作业的资源（处理器和内存）只能被 mapper 和 reducer 使用，而 YARN 为 Hadoop 应用程序提供了更通用的资源访问。因此，专用工具不再需要分解为一系列 MapReduce 作业，可以变得更复杂。通过泛化集群管理，可以扩展最初设想的 MapReduce 编程模型，以包括新的抽象和操作。

Spark 是应运而生的第一个快速、通用的分布式计算范式，并且因其速度和适应性而迅速得到了普及。Spark 主要通过名为弹性分布式数据集（resilient distributed dataset，RDD）的新数据模型实现高速运行。该数据模型在计算时存储在内存中，从而避免了昂贵的中间

磁盘写操作。它还利用了 DAG 执行引擎优化计算，特别是迭代计算，这对于优化算法和机器学习等数据理论任务来说至关重要。在速度方面的优势使得 Spark 能以交互方式进行访问（就像访问 Python 解释器一样），使用户成为计算任务的一部分，并支持以前不可能实现的大数据集探索，让数据科学家能更轻松地使用集群。

因为有向无环图通常用于描述数据流中的步骤，所以在讨论大数据处理时经常会使用 DAG 这一术语。DAG 有向，是因为一个或多个步骤接着前一个步骤；无环，是因为单个步骤不重复。当数据流被描述为 DAG 时，它消除了大代价的同步，并且使得并行应用程序更容易构建。

本章将介绍 Spark 和 RDD，也是讲解使用 Hadoop 进行分析的基础知识的最后一章。因为 Spark 实现了数据科学家熟悉的许多应用程序（例如 DataFrame、交互式 notebook 和 SQL），所以建议 Hadoop 新手用户首选 Spark 与集群交互，至少初期要这样做。为此，我们将描述 RDD，通过 `pyspark` 探索如何在命令行中使用 Spark，然后演示如何使用 Python 编写 Spark 应用程序，并将它们作为 Spark 作业提交到集群中。

4.1　Spark基础

Apache Spark 是一个集群计算平台，为类似于 MapReduce 模型的分布式编程提供了一个 API，但被设计用于快速的交互式查询和迭代算法。[1] 它主要通过在集群节点的内存中缓存计算所需的数据来实现高速运行。在内存中进行集群计算使 Spark 可以运行迭代算法，因为程序可以为数据创建检查点并引用回它，避免从磁盘重新加载。此外，它支持极快速的交互式查询和流式数据分析。因为 Spark 与 YARN 兼容，所以它可以在现有的 Hadoop 集群上运行并访问任何 Hadoop 数据源，包括 HDFS、S3、HBase 和 Cassandra。

还有一点很重要，Spark 的设计从根本上支持大数据应用程序和数据科学任务。Spark API 不仅支持 `map` 和 `reduce`，还提供了许多强大的分布式抽象。这些抽象同样与函数式编程相关，包括 `sample`、`filter`、`join` 和 `collect` 等。此外，虽然 Spark 是用 Scala 实现的，但是 Scala、Java、R 和 Python 的编程 API 使得许多数据科学家能更轻松地使用 Spark，并能够快速且充分地利用 Spark 引擎。

为了理解这种转变，先来看看 MapReduce 在迭代算法方面的局限性。迭代算法对数据块多次应用相同的运算，直到得到预期的结果。例如，像梯度下降这样的优化算法是迭代的——给定某个目标函数（如线性模型），目标是优化该函数的参数，最小化误差（模型的预测值和数据的实际值之间的差）。算法将拥有一组参数的目标函数应用于整个数据集并计算误差，再根据计算得到的误差（沿误差曲线下降）略微修改函数的参数。重复该过程（即迭代），直到误差小于某个阈值或者直到达到最大迭代次数。

这种基本技术是许多机器学习算法（特别是有监督学习）的基础。在有监督学习中，正确答案是提前已知的，可以用它来优化决策空间。为了使用 MapReduce 编程实现这种类型的算法，目标函数的参数必须映射到数据集中的每个实例，并计算和规约误差。在 reduce 阶

注 1：《Spark 快速大数据分析》，Holden Karau、Andy Konwinski、Patrick Wendell、Matei Zaharia 著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。

段结束之后，参数将被更新并馈送到下一个 MapReduce 作业。可以将误差计算和作业更新链接在一起来实现这一步。然而，每个作业都必须从磁盘读取数据，并将误差写回磁盘，这将导致明显的 I/O 延迟。

相反，Spark 在应用程序运行期间将尽可能多的数据集保存在内存中，从而防止在迭代之间重新加载数据。因此，Spark 程序员不是简单地指定 map 步骤和 reduce 步骤，而是在执行某个需要协调的动作（如规约或写入磁盘）之前，指定一系列将应用于输入数据的数据流转换。因为数据流可以通过 DAG 来描述，所以 Spark 的执行引擎提前知道了如何在集群上分发计算并管理计算的细节，这与 MapReduce 抽象分布式计算的方式相似。

通过结合无环数据流和内存计算，Spark 能达到非常快的速度，特别是当集群大到足以容纳内存中所有数据时。事实上，通过增加集群的大小，使内存可以容纳整个非常大的数据集，Spark 可以支持**交互式**使用——使用户成为正在集群上运行的分析进程的关键参与者。随着 Spark 的发展，用户交互的概念成为其分布式计算模型的基础。事实上，很可能就是因为这个原因，Spark 才支持这么多语言。

Spark 的通用性也意味着可以用它构建更高级的工具，用于实现类 SQL 的计算、图形处理和机器学习算法，乃至交互式 notebook 和数据框——这些都是为数据科学家所熟知的工具，但是是在集群环境中实现的。在介绍 Spark **如何**实现通用分布式计算的细节之前，先来了解一下它有哪些工具。

4.1.1 Spark栈

Spark 是一种通用的分布式计算抽象，可以在独立模式下运行。但是 Spark 只专注于**计算**而不关心**数据存储**，因此通常在实现了数据仓储和集群管理工具的集群中运行。本书主要关注 Hadoop（尽管也有 Apache Mesos 和 Amazon EC2 上的 Spark 发行版）。当使用 Hadoop 构建 Spark 时，它使用 YARN 通过 `ResourceManager` 来分配和管理集群资源，如处理器和内存。正因如此，Spark 可以访问所有 Hadoop 数据源，例如 HDFS、HBase、Hive，等等。

Spark 通过 Spark Core 模块将其主要的编程抽象暴露给开发人员。此模块包含基本功能和常规功能，包括定义 RDD 的 API。我们将在下一节中更详细地介绍 RDD，它是所有 Spark 计算的基本功能。Spark 构建于这个核心之上，为各种数据科学任务实现与 Hadoop 交互的专用库，如图 4-1 所示。

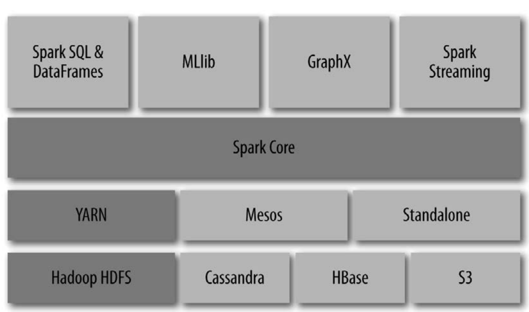


图 4-1：Spark 是一个计算框架，旨在利用集群管理平台（如 YARN）和分布式数据存储（如 HDFS）

因为组件库未集成到通用计算框架中，所以 Spark Core 模块非常灵活，允许开发人员以不同的方法轻松解决类似的用例。例如，将 Hive 迁移到 Spark 可以为现有用户提供一条简单的迁移路径；GraphX 基于以顶点为核心的图计算模型 Pregel，但其他利用 gather、apply、scatter（GAS）风格计算的图形库可以很轻松地用 RDD 实现。这种灵活性意味着 Spark 不仅可以用来开发专业工具，同时也可以帮助新用户快速上手已经存在的 Spark 组件。

Spark 主要包含如下组件。

Spark SQL
: 起初用于与 Spark 进行交互的 API，通过 HiveQL（SQL 的 Apache Hive 变体）实现，你现在也仍然可以通过这个库直接访问 Hive。不过这个库也在不断升级，提供更常规、更结构化的数据处理抽象——DataFrame。DataFrame 本质上是组织成列数据的分布式集合，概念上类似于关系数据库中的表。

Spark Streaming
: 对无界数据流实现实时处理和操作。虽然有许多用于处理实时数据的库（例如 Apache Storm），但是 Spark Streaming 使程序能够像处理一般的 RDD 一样处理实时数据。

MLlib
: 一个常用的机器学习算法库，使用 RDD 上的 Spark 操作实现。该库包含可扩展的学习算法（例如分类、回归等），这些算法需要进行大型数据集的迭代运算。以前人们选择使用的大数据机器学习库是 Mahout 库，以后将转而使用 Spark。

GraphX
: 算法和工具集合，用于操作图形、执行并行图形的操作和计算。GraphX 扩展了 RDD API，囊括了操作图形、创建子图和访问路径中所有顶点的操作。

这些组件与 Spark 编程模型结合，提供了大量与集群资源交互的方法。很可能就是因为这种完整性，Spark 才在分布式分析中如此流行。不需要学习多个工具，基本的 API 在组件中保持不变，而且组件无须额外安装便可轻松访问。这种丰富性和一致性源于 Spark 中的主要编程抽象，即弹性分布式数据集（RDD），到目前为止它已被多次提及，下一节将对它进行更详细的探讨。

4.1.2　RDD

在第 2 章中，我们将 Hadoop 描述为一个分布式计算框架，涉及两个主要问题：如何在集群中分发数据，以及如何分发计算。分布式数据存储问题涉及数据的高可用性（将数据放在它被处理的地方）、可恢复性和持久性。分布式计算意在通过将大型计算或任务分解成更小的独立计算来提高计算的性能（速度），这些计算可以同时（并行）运行，然后聚合得到最终结果。因为每个并行计算在集群中单独的节点或计算机上运行，所以分布式计算框架需要为整个计算提供一致性、正确性和容错保证。Spark 不处理分布式数据存储，而是依靠 Hadoop 提供此功能，因此通过一个被称为弹性分布式数据集的框架专注于提供可靠的分布式计算。

RDD 本质上是一种编程抽象，表示跨机器分区的对象的只读集合。RDD 可以根据转换过

程（lineage）重建（因此是容错的），通过并行操作访问，从分布式存储（例如 HDFS 或 S3）读取和写入，以及高速缓存在 worker 节点的内存中以快速重用——这一点最为重要。如前所述，这种内存缓存功能使速度大幅提高，并提供了机器学习所需的**迭代计算**以及以用户为中心的**交互式分析**。

RDD 使用函数式编程结构体进行操作，函数式编程结构体包括 `map` 和 `reduce`，并在其基础上进行扩展。程序员通过从输入源加载数据，或转换现有集合来创建新的 RDD。RDD **转换过程**（lineage）主要由应用于 RDD 转换的历史定义，并且因为 RDD 的对象集合是**不可变的**（不能直接修改），所以转换可以重新应用于部分或整个集合，以便从故障中恢复。因此，Spark API 本质上是创建、转换和导出 RDD 操作的集合。

> Spark 中的故障恢复与 MapReduce 的区别很大。在 MapReduce 中，数据在每个临时处理步骤之间是作为 sequence 文件（保存带类型的键值对的二进制平面文件）写入磁盘的。因此，进程在 map、shuffle 和 sort、reduce 之间拉取数据。如果一个进程失败，那么另一个进程就开始拉取数据。在 Spark 中，对象集合存储在内存中。通过保留 RDD 部分的早期检查点或缓存版本，RDD 转换过程（lineage）可用于重建部分或全部集合。

因此，基本编程模型描述了如何通过编程操作创建和修改 RDD。有两种类型的操作可以应用于 RDD，分别是**转换**和**动作**。将转换操作应用于现有 RDD 可以创建新的 RDD，例如对 RDD 应用过滤操作可以生成包含过滤出来的值的较小 RDD。然而，动作才会真正将结果返回给 Spark 驱动程序，协调或聚合 RDD 中的所有分区。在这个模型中，`map` 是一种转换操作，因为一个函数被传递给存储在 RDD 中的每个对象，并且该函数的输出映射到一个新的 RDD；而像 `reduce` 这样的聚合是一个动作操作，因为 `reduce` 需要（根据键）对 RDD 进行重新分区，并计算和返回某个聚合值，如和或平均值。Spark 中大多数动作的设计初衷都只是为了输出——返回单个值或小的值列表，或者将数据写回分布式存储。

Spark 的另一个好处是，它会“延迟”应用转换操作，即向集群提交作业以执行作业之前，检查完整的转换序列和一个动作。这种延迟执行机制带来了明显的存储和计算优化，因为它允许 Spark 建立数据转换过程（lineage）并评估完整的转换链，以便只计算结果所需的数据。例如，如果对 RDD 运行 `first()` 动作，Spark 将不会读取整个数据集，并只返回第一个匹配行。

4.1.3 使用RDD编程

Spark 应用程序的编写与之前在 Hadoop 上实现的其他数据流框架很像。代码被写在驱动程序（driver program）中，提交时在驱动程序所在机器上被延迟评估。一旦遇到动作，驱动程序代码就被分发到集群上，由 worker 节点在各自的 RDD 分区上执行该代码。然后结果被发送回驱动程序以进行聚合或汇编。如图 4-2 所示，驱动程序通过将来自 Hadoop 数据源的数据集并行化，创建一个或多个 RDD，应用操作来转换 RDD，然后对经过转换的 RDD 调用某个动作以检索输出。

术语**并行化**已经出现好几次了，有必要来解释一下。RDD 是分区的数据集合，允许程序员**并行地**对整个集合应用操作。正是分区支持了并行化，而且分区本身也是数据列表中计算的边界，其中数据存储在不同的节点上。因此，“并行化”是一种行为，它将数据集分区，并将数据的每个部分发送到将对其执行计算的节点。

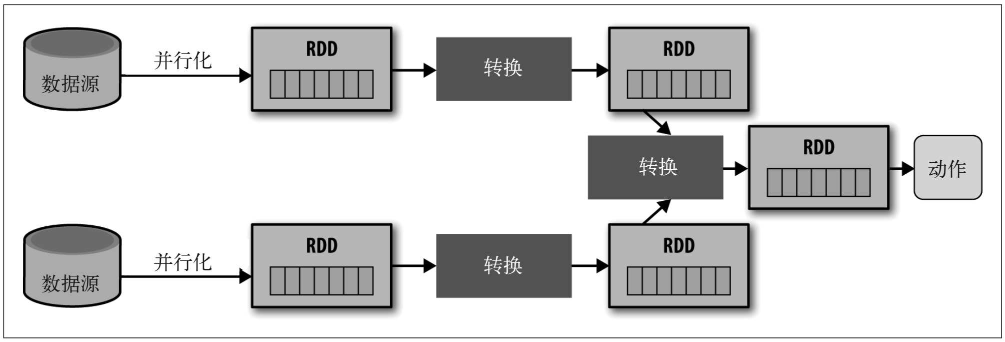


图 4-2：一个将集群中的数据集并行化（分区）为 RDD 的典型 Spark 应用程序

Spark 编程的典型数据流序列如下所示。

(1) 通过访问存储在磁盘（例如 HDFS、Cassandra、HBase 或 S3）上的数据、并行化某个集合、转换现有 RDD 或缓存来定义一个或多个 RDD。缓存是 Spark 的一个基本过程，即在节点内存中存储 RDD，用于在计算过程中快速访问。

(2) 通过将闭包（这里指不依赖于外部变量或数据的函数）传递到 RDD 的每个元素，来对 RDD 执行操作。Spark 提供了除 `map` 和 `reduce` 之外的许多高级算子。

(3) 对生成的 RDD 使用聚合动作（例如计数、收集、保存等）。动作将启动集群上的计算，因为在计算聚合之前无法进行任何计算。

简单解释一下变量和闭包，这是两个在 Spark 中容易混淆的概念。当 Spark 在 worker 节点上运行闭包时，闭包中使用的所有变量都将被复制到该节点，但在该闭包的局部范围内维护。如果需要外部数据，Spark 提供了两种类型的共享变量——**广播变量**和**累加器**，所有的 worker 节点都可以通过受限方式与它们交互。广播变量被分发给所有 worker 节点，但是只读的，并且通常作为查找表或禁用词列表使用。累加器是一个变量，worker 节点可以“累加”（满足结合律）它，通常用作计数器。这些数据结构类似于 MapReduce 中的分布式缓存和计数器，并且发挥着类似的作用。但是，由于 Spark 支持一般的进程间通信，所有这些数据结构可以用于更广泛的应用程序。

闭包是一种炫酷的函数式编程技术，使分布式计算得以实现。它们提供词法作用域名称绑定，这基本就意味着闭包是包括自己的独立数据环境的函数。由于存在这种独立性，闭包的运算不使用外部信息，因此是可并行化的。闭包在日常编程中越来越常见，通常用作回调。在其他语言中，也可能被称为**块**或**匿名函数**。

尽管以下内容包含了如何使用 Spark 来执行分布式计算的示例，但 Spark 开发人员可用的转换和动作的完整指南不在本书的讨论范围之内。有关 Spark 所支持的转换和动作的完整列表以及使用文档，可以在 Spark Programming Guide（http://spark.apache.org/docs/latest/programming-guide.html）中找到。下一节将讨论如何使用 Spark 以交互方式在命令行上使用转换和动作，而免于编写完整的程序。

Spark 执行机制

以下内容是 Spark 执行机制的简要说明。Spark 应用程序本质上是独立运行的进程的集合，由驱动程序中的 SparkContext 进行协调。该上下文将与某个分配系统资源的集群管理器（例如 YARN）连接。集群中的每个 worker 节点都由一个 executor 管理，executor 又由 SparkContext 管理。executor 管理每台机器上的计算、存储和缓存。驱动程序、YARN 和 worker 节点的交互如图 4-3 所示。

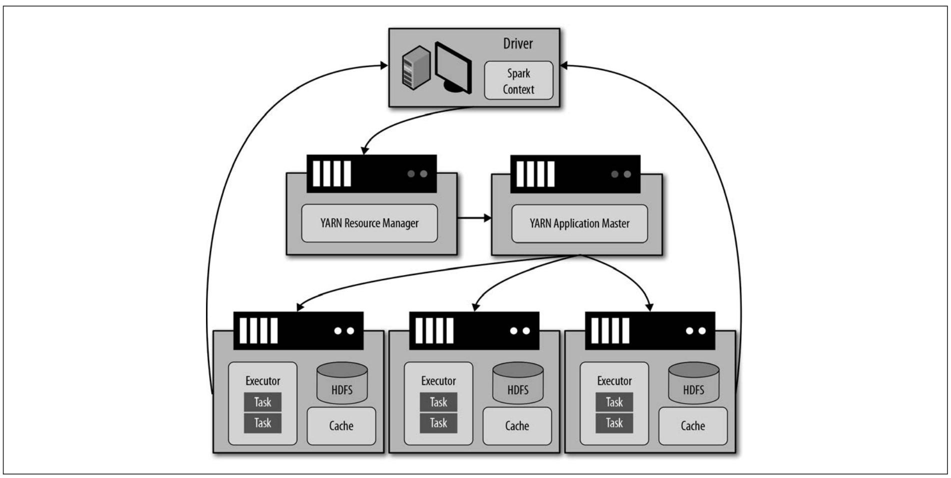


图 4-3：在 Spark 执行模型中，驱动程序是处理的一个重要部分

但要注意，应用程序代码从驱动程序发送到 executor，executor 指定上下文和要运行的各种任务。executor 与驱动程序来回通信以进行数据的共享或交互。驱动程序是 Spark 作业的关键参与者，因此它们应该和集群待在同一个网络上。这不同于 Hadoop 代码——你可以从任何地方将作业提交到 `ResourceManager`，并由 `ResourceManager` 负责作业在集群上的执行。

考虑到这一点，就能明白其实可以通过两种模式将 Spark 应用程序提交到 Hadoop 集群，分别是 `yarn-client` 和 `yarn-cluster`。在 `yarn-client` 模式下，驱动程序在客户端进程内运行。如上所述，`ApplicationMaster` 仅管理作业的进度并请求资源。然而在 `yarn-cluster` 模式下，驱动程序在 `ApplicationMaster` 进程内部运行，因此释放了客户端进程，像传统的 MapReduce 作业一样运行。如果程序员想获取即时结果或想以交互模式运行，可以使用 `yarn-client` 模式；而对于时间运行长或不需要用户干预的作业，使用 `yarn-cluster` 模式更为合适。

4.2　基于PySpark的交互性Spark

Spark 处理起可以放入集群内存的数据集非常快，足以支持数据科学家在实现了 Python REPL（read-evaluate-print loop，读取、评估、打印循环）的交互式 shell 中交互并探索大数据。Spark 中的交互式 shell 叫 pyspark。这种交互方式类似于在 Python 解释器中与本地 Python 代码交互、在命令行中编写命令并接收 stdout 的输出（还有 Scala 和 R 的交互式 shell）。这种类型的交互还支持交互式 notebook，在 Spark 环境中设置 iPython 或 Jupyter notebook 也非常容易。

本节将开始研究如何在 pyspark 中使用 RDD，因为这是启用 Spark 最简单的方法。为了运行交互式 shell，你需要定位 pyspark 命令，该命令位于 Spark 库的 bin 目录。和 $HADOOP_HOME（一个指向系统上 Hadoop 库的位置的环境变量）类似，你也应该配置一个 $SPARK_HOME。Spark 无须配置即可运行，因此只需下载适用于系统的 Spark 就足够了。将 $SPARK_HOME 替换为下载路径（或设置你的环境）就可以运行交互式 shell，如下所示：

```
hostname $ $SPARK_HOME/bin/pyspark
[... snip ...]
>>>
```

PySpark 使用本地 Spark 配置自动创建了一个 SparkContext。它通过 sc 变量将自己暴露给终端。来创建第一个 RDD 吧：

```
>>> text = sc.textFile("shakespeare.txt")
>>> print text
shakespeare.txt MappedRDD[1] at textFile at NativeMethodAccessorImpl.java:-2
```

textFile 方法将莎士比亚全集（http://bit.ly/16c7kPV）从本地磁盘加载到名为 text 的 RDD 中。如果你检查 RDD，就会看到它是一个 MappedRDD，并且文件的路径是当前工作目录（你系统中要传入 shakespeare.txt 文件的正确路径）的相对路径。与第 2 章中的 MapReduce 示例类似，先来转换这个 RDD，以便计算分布式计算中的“Hello, World”，并使用 Spark 实现单词计数应用程序：

```
>>> from operator import add
>>> def tokenize(text):
...     return text.split()
...
>>> words = text.flatMap(tokenize)
```

我们导入了算子 add，一个可以用作加法闭包的命名函数，等会儿就要用到它。第一件要做的事是将文本拆分成单词：创建一个名为 tokenize 的函数（其参数是某个文本片段），通过简单地使用空格拆分文本返回该文本中的令牌（单词）列表。然后，通过应用 flatMap 算子并将 tokenize 闭包传递给它，转换 text RDD 以创建一个叫作 words 的新 RDD。

此时，我们有了一个类型为 PythonRDD 的 RDD words。你可能已经注意到，输入这些命令之后立即就产生了结果（尽管有一个意料之中的轻微处理延迟，因为要将莎士比亚全集拆分成单词）。因为 Spark 执行延迟评估，所以处理的执行（读取数据集、跨进程的分区，以及将 tokenize 函数映射到集合）还未发生。相反，PythonRDD 描述了创建此 RDD 的前提，并且在描述中还维护了数据到单词的转换过程。

因此，我们可以继续对这个 RDD 应用转换，免于经历漫长的分布式执行过程，而且还可能是个存在错误或非最佳的执行过程。如第 2 章所述，接下来的步骤是将每个单词映射到一个键值对，其中键是单词，值是 1，然后使用 reducer 对每个键的 1 进行求和。首先应用 map：

```
>>> wc = words.map(lambda x: (x,1))
>>> print wc.toDebugString()
(2) PythonRDD[3] at RDD at PythonRDD.scala:43
| shakespeare.txt MappedRDD[1] at textFile at NativeMethodAccessorImpl.java:-2
| shakespeare.txt HadoopRDD[0] at textFile at
NativeMethodAccessorImpl.java:-2
```

这次不使用命名函数，而是使用匿名函数（在 Python 中使用 lambda 关键字）。这行代码将 lambda 映射到 words 中的每个元素。因此，每个 x 是一个单词，并且该单词将被匿名闭包转换成一个元组 (单词 ,1)。为了检查到目前为止的转换过程（lineage），可以使用 toDebugString 方法来查看 PipelinedRDD 是如何转换的。然后应用 reduceByKey 动作获取单词计数，将它们写入磁盘：

```
>>> counts = wc.reduceByKey(add)
>>> counts.saveAsTextFile("wc")
```

一旦调用 saveAsTextFile 动作，分布式作业就会启动。当作业“在集群中”（或者仅作为本地机器上的多个进程）运行时，你会看到大量的 INFO 语句。如果退出解释器，就会在当前工作目录中看到一个名为 wc 的目录：

```
hostname $ ls wc/
_SUCCESS   part-00000 part-00001
```

每个 part 文件表示最终 RDD 的一个分区，最终 RDD 由计算机上的各种进程计算并保存到磁盘。如果对一个 part 文件使用 head 命令，则会看到单词计数对的元组：

```
hostname $ head wc/part-00000
(u'fawn', 14)
(u'Fame.', 1)
(u'Fame,', 2)
(u'kinghenryviii@7731', 1)
(u'othello@36737', 1)
(u'loveslabourslost@51678', 1)
(u'1kinghenryiv@54228', 1)
(u'troilusandcressida@83747', 1)
(u'fleeces', 1)
(u'midsummersnightsdream@71681', 1)
```

请注意，在 MapReduce 作业中，由于 map 和 reduce 之间有 shuffle 阶段和 sort 阶段，所以键会被排序。但因为所有 executor 都可以相互通信，所以 Spark 进行 reduce 时不会对重新分区排序。因此，前面的输出不会按照字母顺序排列。不过，由于用 reduceByKey 算子聚合了 counts RDD，所以即使没有排序，仍然能保证每个键在所有 part 文件中仅出现一次。如果需要排序，可以使用 sort 算子确保所有键在写入磁盘之前都已被排序。

4.3 编写Spark应用程序

使用 Python 编写 Spark 应用程序与在交互式控制台中使用 Spark 很像，因为 API 是相同的。但是你不需要在交互式 shell 中输入命令，而是需要创建一个完整的、可执行的驱动程序并将其提交到集群。这涉及一些在 `pyspark` 中自动处理的内务任务，包括获取 `SparkContext` 的访问，这是由 shell 自动加载的。

因此，许多 Spark 程序都是简单的 Python 脚本。它包含一些数据（共享变量），定义用于转换 RDD 的闭包，并描述 RDD 转换和聚合的分步执行计划。使用 Python 编写 Spark 应用程序的基本模板如下所示：

```
## Spark应用程序,使用spark-submit执行

## 导入
from pyspark import SparkConf, SparkContext

## 共享变量和数据
APP_NAME = "My Spark Application"

## 闭包函数
## 主要功能
def main(sc):
    """
    这里描述RDD转换和动作
    """
    pass

if __name__ == "__main__":
    # 配置Spark
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc   = SparkContext(conf=conf)

    # 执行主要功能
    main(sc)
```

此模板展示了 Python 语言的 Spark 应用程序自上而下的结构：`import` 让各种 Python 库以及 Spark 组件（例如 GraphX 和 SparkSQL）可用于分析。为了调试和日志记录，共享数据和变量被指定为模块常量，包括在 Web UI 中使用的应用程序名称。为了便于调试，可以让驱动程序包含特定于作业的闭包或自定义算子，这些闭包或自定义算子也可以导入到其他 Spark 作业中。最后，`main` 方法定义转换和聚合 RDD 的分析方法，该 `main` 方法作为驱动程序运行。

资深 Python 程序员应该会注意到这里使用了 `if __name__ == '__main__'`（通常被称为 `ifmain`）语句，其中 Spark 配置和 `SparkContext` 被定义并传递给 `main` 函数。通过 `ifmain` 可以轻松地将驱动程序代码导入到其他 Spark 上下文，无须创建新的上下文或配置、执行作业（在导入时，名称不是 `__main__`）。具体来说，Spark 程序员通常会将代码从应用程序导入到 iPython、Jupyter notebook 或 `pyspark` 交互式 shell 中，以便在对较大的数据集运行作业之前进行分析。

驱动程序定义了 Spark 执行过程的方方面面，例如程序员可以在代码中使用 sc.stop() 或 sys.exit(0) 停止或退出程序。这样的控制也可以扩展到执行环境——在这个模板中，Spark 集群的配置 local[*] 通过 setMaster 方法硬编码到 SparkConf 中，这告诉 Spark 在本地机器上运行尽可能多的进程（多进程，但不是分布式计算）。虽然你可以在命令行使用 spark-submit 来指定 Spark 执行的位置，但是驱动程序通常基于使用 os.environ 的环境变量来进行选择。因此，在开发 Spark 作业（例如使用 DEBUG 变量）时，作业可以在本地运行；但是在生产环境中，作业在集群的较大数据集上运行。

编写 Spark 应用程序肯定与编写 MapReduce 应用程序不同，因为转换和动作提供了灵活性，以及更灵活的编程环境。在下一节中，我们将看到一个完整的分析，它利用驱动程序的中心性来计算集群中的数据，从而创建一个可视化输出。

使用Spark可视化航班延误

第 3 章探讨了如何根据美国交通部的准点航班数据集（http://bit.ly/1Dz76xB），使用 Hadoop Streaming 和 MapReduce 计算每个机场的平均航班延误时间。这种解析 CSV 文件并执行聚合的计算是 Hadoop 的一个极其常见的用例，主要由于 CSV 数据很容易从关系数据库导出。该数据集记录了美国所有国内航班的起飞时间、到达时间及其延误时间。有趣的是，虽然单月的数据很容易计算，但整个数据集因为数据量太大，所以必须使用分布式计算。

本例将使用 Spark 来执行数据集的聚合，具体来说，是确定哪些航空公司在 2014 年 4 月的总延误时间最长。由于 Spark 的 Python API 更加灵活，所以可以具体看看能使用哪些稍微高级些（和 Pythonic）的技术。此外，我们将通过 matplotlib 将结果拉取回来并在驱动程序机器上显示可视化图表，从而证明驱动程序对计算有多么重要（如果用传统的 MapReduce 执行此任务，则需要两个步骤）。

为了了解 Spark 应用程序的结构，并学习上一节中描述的模板的实际用法，我们将从宏观上把握程序的整体结构而忽略其细节：

```
## 导入
import csv
import matplotlib.pyplot as plt

from StringIO import StringIO
from datetime import datetime
from collections import namedtuple
from operator import add, itemgetter
from pyspark import SparkConf, SparkContext

## 模块常数
APP_NAME = "Flight Delay Analysis"
DATE_FMT = "%Y-%m-%d"
TIME_FMT = "%H%M"

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')
Flight   = namedtuple('Flight', fields)
```

```python
## 闭包函数
def parse(row):
    """
    解析一行并返回一个命名元组
    """
    pass

def split(line):
    """
    使用csv模块分割行的函数
    """
    pass

def plot(delays):
    """
    显示航空公司总延误状况的柱状图
    """
    pass

## 主要功能
def main(sc):
    """
    描述数据集上应用的转换和动作，
    然后使用matplotlib绘制输出结果的可视化
    """
    pass

if __name__ == "__main__":
    # 配置Spark
    conf = SparkConf().setMaster("local[*]")
    conf = conf.setAppName(APP_NAME)
    sc   = SparkContext(conf=conf)

    # 执行主要功能
    main(sc)
```

这段代码虽然很长，但却是现实中 Spark 程序结构的一个良好概览。`import` 说明标准库工具和第三方库（如 `matplotlib`）通常混合使用。与 Hadoop Streaming 一样，任何不属于标准库的第三方代码都必须预先安装在集群上，或随作业一起提供。对于只需要在驱动程序上执行，而不需要在 executor 中执行的代码（如 `matplotlib`），可以使用 `try/except` 块捕获 `ImportError`。

与 Hadoop Streaming 一样，任何不属于 Python 标准库的第三方 Python 依赖都必须预先安装在集群中的**每个**节点上。但是，与 Hadoop Streaming 不同的是，Spark 有两个上下文（即驱动程序上下文和 executor 上下文），这允许一些重量级库（特别是可视化库）可以只安装在驱动程序机器上——只要它们不被在集群上运行的 Spark 运算使用的闭包使用就可以。如果想要预防错误，可以使用 `try/except` 块包裹 `import` 语句来捕获 `ImportError`。

然后，应用程序定义了一些可配置的数据，包括用于解析 `datetime` 字符串的日期和时间格式以及应用程序名称。它还创建了专用的 `namedtuple` 数据结构，以便从输入数据创建轻量级、可访问的行。此信息应该可为全部 executor 所用，但是它又足够轻量，所以不需要广播变量。接下来，定义处理函数 `parse`、`split` 和 `plot`，再使用 `SparkContext` 定义航空公司数据集上动作和转换的 `main` 函数。最后，`ifmain` 配置 Spark 并执行 `main` 函数。

有了这个完整的高级概览之后，再来深入了解一下代码的细节。从定义主要 Spark 操作和分析方法的 `main` 方法开始：

```
## 主要功能
def main(sc):
    """
    描述在数据集上应用的转换和动作，
    然后使用matplotlib绘制输出结果的可视化
    """

    # 加载航空公司查找字典
    airlines = dict(sc.textFile("ontime/airlines.csv").map(split).collect())

    # 将查找字典广播到集群
    airline_lookup = sc.broadcast(airlines)

    # 读取CSV数据到一个RDD
    flights = sc.textFile("ontime/flights.csv").map(split).map(parse)

    # 映射总延误时间到航空公司(使用广播变量进行关联)
    delays = flights.map(lambda f: (airline_lookup.value[f.airline],
                                    add(f.dep_delay, f.arv_delay)))

    # 航空公司当月总延误时间
    delays = delays.reduceByKey(add).collect()
    delays = sorted(delays, key=itemgetter(1))

    # 驱动程序提供输出
    for d in delays:
        print "%0.0f minutes delayed\t%s" % (d[1], d[0])

    # 显示延误时间柱状图
    plot(delays)
```

第一个任务是从磁盘加载两个数据源：航空公司代码对应航空公司名称的查找表，以及航班数据集。数据集 airlines.csv 是一个小跳转表，允许通过航空公司代码获取完整的航空公司名称。不过由于这个数据集非常小，所以不必执行两个 RDD 的分布式连接，而是将此信息存储为 Python 字典，并使用 `sc.broadcast` 将其广播到集群中的每个节点。`sc.broadcast` 将本地 Python 字典转换为广播变量。

下面来说说这个广播变量的创建和执行。首先，从本地磁盘上的文本文件 airlines.csv（注意它的相对路径）创建一个 RDD。之所以需要创建 RDD，是因为这些数据可能来自 Hadoop 数据源，而该数据源可能由其位置的 URI（例如，HDFS 数据使用 hdfs://、S3 使用 s3:// 等）指定。请注意，如果这个文件只在本地机器上，则不需要将其加载到 RDD 中。然后，将 `split` 函数映射到数据集中的每个元素，如上所述。最后，将 `collect` 动作应用

于 RDD，它将数据作为 Python 列表从集群返回到驱动程序。因为应用了 `collect` 动作，所以当这行代码执行时，作业将被发送到集群以加载、拆分 RDD，并将上下文返回到驱动程序：

```
def split(line):
    """
    使用csv模块分割行的函数
    """
    reader = csv.reader(StringIO(line))
    return reader.next()
```

`split` 函数使用 `csv` 模块解析每行文本——使用 `StringIO` 创建一个带文本行的类似文件的对象，然后将其传递给 `csv.reader`。因为只有一行文本，所以可以简单地返回 `reader.next()`。虽然这种 CSV 解析方法看似相当重量级，但是它让我们能更轻松地处理分隔符、转义和 CSV 处理中的其他细枝末节。对于较大的数据集，通过类似的方法，使用 `sc.wholeTextFiles` 处理大量被分割为 128MB 的块（即 HDFS 上的块大小和副本大小）的整个 CSV 文件：

```
def parse(row):
    """
    解析行并返回一个命名元组
    """

    row[0]  = datetime.strptime(row[0], DATE_FMT).date()
    row[5]  = datetime.strptime(row[5], TIME_FMT).time()
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row[7], TIME_FMT).time()
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

接下来，`main` 函数加载更大的 flights.csv，后者需要使用 RDD 以并行方式计算。分割 CSV 行之后，将 `parse` 函数映射到 CSV 行，这可以将日期和时间转换为 Python 的日期和时间，并适当地转换浮点数。此函数的输出是一个名为 `Flight` 的 `namedtuple`，在应用程序的模块常量部分定义。命名元组是包含记录信息的轻量级数据结构，允许通过名称（例如 `flight.date`）而不是位置（例如 `flight[0]`）来访问数据。和普通的 Python 元组一样，命名元组也不可变，因此可以放心地将它们用于处理程序，数据不会被修改。此外，与字典相比，它们占用的内存更小，处理效率也更高。因此，命名元组在内存尤显珍贵的大数据应用程序（如 Spark）中具有显著的优势。

有了 `Flight` 对象的 RDD 后，我们要做的最后一个转换是映射一个匿名函数，将 RDD 转换成一系列键值对，其中键是航空公司的名称，值是到达和起飞的延误时间之和。目前除了创建航空公司字典之外，还没有对该集合执行过任何操作。不过一旦开始使用 `reduceByKey` 动作和 `add` 算子计算每个航空公司的延误时间之和，该作业就将在集群中执行，运算结果将被收集回驱动程序。

此时，集群计算已经完成，驱动程序以串行方式继续运行。延误时间按照延误程度在客户端程序的内存中进行排序。请注意，之所以能这么做，原因和创建航空公司查找表作为广

播变量的原因相同：航空公司的数量很少，在内存中排序的效率更高。但如果这个 RDD 非常大，就可以使用 rdd.sort 进行分布式排序。最后，我们没有将结果写入磁盘，而是将输出打印到控制台。如果数据集较大，则可以使用 rdd.first 动作来获取前 n 个项目，而不用打印整个数据集；或者可以使用 rdd.saveAsTextFile 将数据写回到本地磁盘或 HDFS。

最后，因为数据在驱动程序中可用，所以可以使用 matplotlib 可视化结果，如下所示：

```
def plot(delays):
    """
    显示航空公司总延误状况柱状图
    """
    airlines = [d[0] for d in delays]
    minutes  = [d[1] for d in delays]
    index    = list(xrange(len(airlines)))

    fig, axe = plt.subplots()
    bars = axe.barh(index, minutes)

    # 在右侧添加总分钟数
    for idx, air, min in zip(index, airlines, minutes):
        if min > 0:
            bars[idx].set_color('#d9230f')
            axe.annotate(" %0.0f min" % min, xy=(min+1, idx+0.5), va='center')
        else:
            bars[idx].set_color('#469408')
            axe.annotate(" %0.0f min" % min, xy=(10, idx+0.5), va='center')

    # 设置tick
    ticks = plt.yticks([idx+ 0.5 for idx in index], airlines)
    xt = plt.xticks()[0]
    plt.xticks(xt, [' '] * len(xt))

    # 最小化图表垃圾
    plt.grid(axis = 'x', color ='white', linestyle='-')

    plt.title('Total Minutes Delayed per Airline')
    plt.show()
```

希望此示例能说明集群和驱动程序的交互过程（发送数据进行分析，然后将结果返回到驱动程序），以及 Python 代码在 Spark 应用程序中起到的作用。要运行此代码（假定你有一个名为 ontime 的目录，其中有这两个 CSV 文件），请使用 spark-submit 命令，如下所示：

```
hostname $ spark-submit app.py
```

因为在 ifmain 中将 master 节点的配置硬编码为了 localhost[*]，所以此命令创建了一个 Spark 作业，它有 localhost 上尽可能多的进程。这个 Spark 作业将使用本地 SparkContext 执行 main 函数中指定的转换和动作：首先，将跳转表加载为 RDD，收集它并将它广播给所有进程；然后，加载航班数据 RDD 并以并行方式计算平均延误时间。

一旦上下文和 collect 的输出返回到驱动程序，就可以使用 matplotlib 来可视化结果了，如图 4-4 所示。最终结果显示，2014 年 4 月，夏威夷航空公司和阿拉斯加航空公司的总延误时间（以分钟为单位）最短，甚至有提前达到，其余大航空公司均有延误。本例的新颖

之处不在于分析的可视化，而是在一步步提交并行执行作业的过程中，在合理的时间内显示结果。因此，这类直接向用户提供按需分析以立即获得有用信息的应用程序变得越来越普遍。

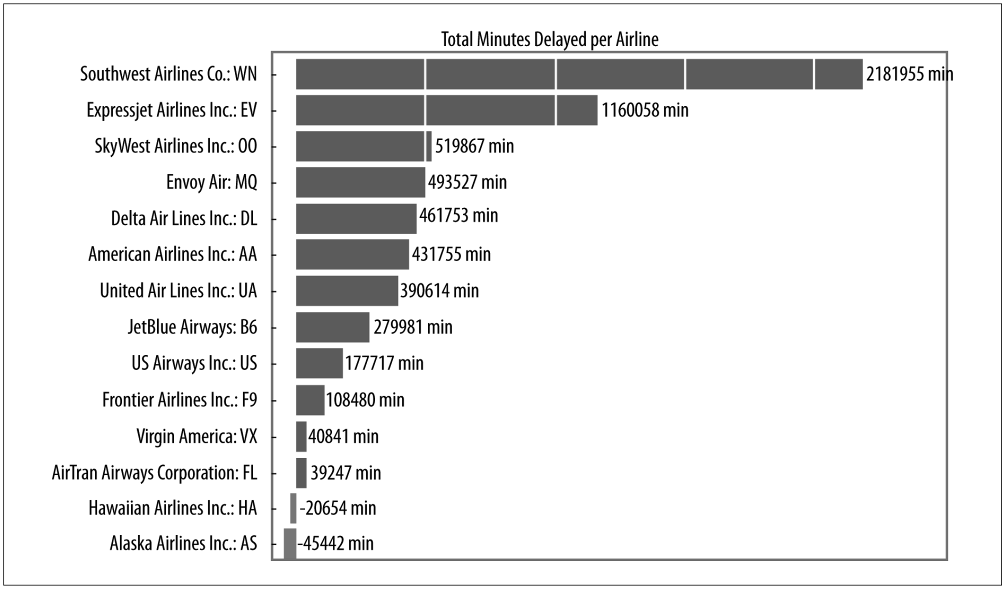


图 4-4：数据集中延误时间最长的航空公司的可视化图

4.4 小结

虽然 Spark 最初旨在突破 MapReduce 在执行迭代算法方面的局限性，但它现在已经发展成为了一个完整的通用分布式计算引擎。Spark 已经应用于大量大数据处理工作负载——它们都使用了 Spark 这一通用引擎，而没有实现专用系统。因为 Spark 比传统的 MapReduce 快 10~20 倍，所以你可能想知道 Spark 在 Hadoop 生态系统中处于什么位置。虽然说 Spark 终将替代 MapReduce 还为时过早，但它已经吸引了相当多采用了 Hadoop 的企业和公司，这些组织需要一个可以使用现有的集群资源执行接近实时计算的平台。但是要记住，至少现在 Spark 还不是 Hadoop 和 MapReduce 的替代品，而应该被认为是它们的扩展，并且可以与 Hadoop 生态系统的其他组成部分并存。

Spark 不解决分布式存储问题（通常 Spark 从 HDFS 获取其数据），但它为分布式计算提供了丰富的函数式编程 API。这种框架建立在**弹性分布式数据集**（RDD）的理念上。RDD 是一种编程抽象，表示分区的对象集合，允许对它们执行分布式操作。RDD 是容错的（弹性部分），还可以存储在 worker 节点的内存中，以便快速重用。内存存储提供更快、更容易表达的迭代算法，还支持实时交互式分析。

因为 Spark 库拥有 Python、R、Scala 和 Java 的 API，以及用于机器学习、流式数据、图形算法和类 SQL 查询的内置模块，所以它已经迅速成为现今最重要的分布式计算框架之一。

当与 YARN 结合时，Spark 用于增强（不是替换）现有的 Hadoop 集群。未来，它将成为大数据的重要组成部分，开辟数据科学探索的新途径。

这一章远不止是 Spark 的完整介绍；相反，它介绍了 Spark 计算框架和弹性分布式数据集，提供了如何与 Spark 交互和编程的思路。因为本书的目标读者是了解 Python 或 R 的数据科学家，所以他们可能觉得 Spark 比 Hadoop Streaming 更适合作分析。在本书余下的内容中，我们将使用 Hadoop Streaming 和 Spark 进行计算，但在大多数情况下——特别是与机器学习相关时——将主要使用 Spark。

第 5 章

分布式分析和模式

MapReduce 和 Spark 让开发人员和数据科学家能轻松进行**数据并行**运算。在这类运算中，数据被分发到多个处理节点同时计算，然后通过 reduce 产生最终输出。而 YARN 提供了简单的**任务并行性**，通过为每个任务分配自由计算资源，支持集群同时执行多个不同的操作。并行缩短了执行单个计算所需的时间，从而支持以每秒数千条记录的速度分析 PB 级的数据集，或处理由多个异构数据源组成的数据集。然而，大多数并行运算（如前述运算）都比较简单。这也带来了一个问题：数据科学家如何进行大规模高级数据分析？

不妨用 Creighton Abrams 的一句话总结一下大规模分析的主要原则："要想吃掉一头大象，也得一口一口来。"单个运算只对数据进行多个微小的处理，而要想得到更有意义的结果，必须将这些运算组成一个被称为**数据流**的分步序列。如果两个运算可以同时进行，则数据流可以分叉（fork）和合并（merge），支持任务和数据并行，但是必须保证序列的数据从输入数据源串行馈送到最终输出。因此，数据流被描述为**有向无环图**（DAG）。如果一种算法、分析或精心设计的计算可以表示为 DAG，则它可以在 Hadoop 上并行化，认识到这一点非常重要。

不幸的是，你很快就会发现，许多算法都不能轻易转换为 DAG，也就不适合这种类型的并行化了。不能被描述为有向数据流的算法有：在整个计算过程中维持或更新单个数据结构的算法（需要一些共享内存），或者依赖中间步骤计算结果的算法（需要中间进程间通信）。引入循环的算法，特别是循环次数不限的迭代算法，也不容易描述为 DAG。

有一些工具和技术可以满足 MapReduce 和 Spark 中循环性、共享内存或进程间通信的需求。但是要利用这些工具，必须将算法重写为分布式形式。比起重写算法，更常采用的是一种技术要求更低但同样有效的方法：设计一种数据流，将输入域分解为适合单个机器内存的较小输出，对输出运行串行算法，然后使用另一个数据流在集群上验证该分析（例如计算误差）。

正是因为这种方法被广泛采用，Hadoop 才被普遍认为是一个释放大数据集潜力的预处理器——它通过每个操作将数据集规约（reduce）成越来越容易管理的块。一种常见做法是，使用 MapReduce 或 Spark 将数据分解到一个可以载入 128GB 内存（高性价比机器的硬件要求）的计算空间中。这个规则通常被称为“最后一英里”（last-mile）计算，因为它将数据从极大的空间移动到足够近的地方（即最后一英里），从而能够进行准确的分析或特定于应用程序的计算。

在本章中，我们将在数据流的上下文中探讨将计算空间缩小或分解为更易于管理的计算空间的并行计算模式。首先，讨论基于键的计算，这是 MapReduce 的需求，对 Spark 也至关重要；接着，学习概要（summarization）、索引（indexing）和过滤（filtering）的模式，这些模式是大多数分解算法的关键部分——在这个上下文中，我们将讨论统计概要、抽样、搜索和分类（binning）的应用；最后，纵览三种回归、分类和聚类风格分析的预处理技术。

> 本章将介绍一些 Hadoop 生态系统中使用的方法。这些方法也被其他项目使用，具体内容将在最后 4 章讨论。本章还将讨论表示为数据流的算法，而第 8 章将接着讨论用于创建数据流的工具，包括高级 API（如 Pig 和 Spark Data Frames）。本章讨论的许多过滤和概要算法更容易表示为结构化查询，第 7 章将具体讨论如何在 Hadoop 上使用 Hive 执行结构化查询。最后，这些章节中的组件（包括 Sckit-Learn 模型的用法）是理解使用 Spark 的 MLlib 进行机器学习的第一步，这将在第 9 章中讨论。

本章还介绍了经常用于数据分析的标准算法，包括统计概要（并行版的“describe”命令）、并行 grep、TF-IDF 和 canopy 聚类。通过这些例子，我们将阐明 MapReduce 和 Spark 的基本机制。

5.1 键计算

要理解数据流实际是如何工作的，第一步就是理解键值对和并行计算之间的关系。在 MapReduce 中，**所有数据**在 map 阶段和 reduce 阶段都被构造为键值对。关键需求主要与 reduce 有关，因为聚合是按键分组的，并行 reduce 需要对**键空间**（换句话说，就是键的值域）进行分区，使每一个 reducer 任务都能收到一个键的所有值。如果没有用于分组的键（这其实很常见），你就可以按单一的键进行 reduce，强制对所有映射值进行 reduce。然而在这种情况下，reduce 阶段无法从并行中受益。

键虽然经常被忽略（特别是在 mapper 中，键仅仅是文档标识符），但是它能让计算在数据集上同时进行。因此，数据流表达了一组值与另一组值之间的关系。这听起来很耳熟，尤其是在更传统的数据管理方式的上下文中，它相当于关系数据库的结构化查询。就像你不会在 PostgreSQL 这样的数据库上运行多个单独的查询来进行不同维度的分析，MapReduce 和 Spark 计算采用了并行执行分组操作，如图 5-1 所示的按键分组的平均值计算。

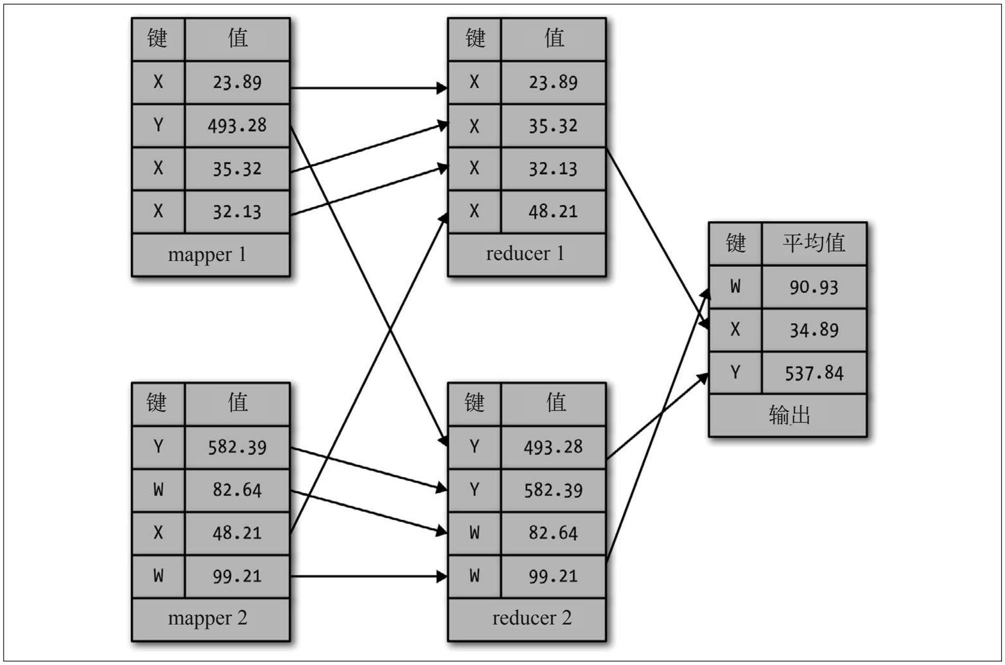


图 5-1：通过将键空间分区到多个 reducer，键支持并行 reduce

此外，键可以保存在数据流的一个阶段中已经被 reduce 的信息，还能自动并行下一步计算所需的结果。这是通过复合键完成的，下一节将讨论这种技术，它表明键不一定是简单的、原始的值。事实上，键对于这些类型的计算非常有用，尽管它们在使用 Spark 的计算中不是必需品（RDD 可以是简单值的集合），但大多数 Spark 应用程序需要它们来进行分析，主要是使用 `groupByKey`、`aggregateByKey`、`sortByKey` 和 `reduceByKey` 动作来收集和 reduce。

5.1.1　复合键

键不一定是简单的原始数据类型，如整型或字符串；相反，它们可以是复合类型或复杂类型，只要是**可散列**（hashable）且**可比较**（comparable）的即可。可比较的类型必须至少能暴露某种用于判断相等的机制（用于 shuffle）和某种排序方法（用于 sort）。一般通过将某种类型映射到一个数值（例如，将月份映射到整数 1~12），或通过词汇顺序来完成比较。Python 中的可散列类型是任意一种不可变类型，最典型的就是元组。但是元组可以包含可变类型（例如一个列表的元组），因此可散列的元组是指由不可变类型组成的元组。像列表和字典这样的可变类型可以转换为不可变的元组。

```
# 将列表转换为元组
key = tuple(['a', 'b', 'c'])

# 将字典转换为由元组构成的元组
key = {'a': 1, 'b': 2}
key = tuple(key.items())
```

使用复合键的方式主要有两种：在多个维度上划分键空间，以及在仅涉及值的计算阶段携带特定于键的信息。思考以下形式的 Web 日志记录：

```
local - - [30/Apr/1995:21:18:07 -0600] "GET 7448.html HTTP/1.0" 404 -
local - - [30/Apr/1995:21:18:42 -0600] "GET 7448.html HTTP/1.0" 200 980
remote - - [30/Apr/1995:21:22:56 -0600] "GET 4115.html HTTP/1.0" 200 1363
remote - - [30/Apr/1995:21:26:29 -0600] "GET index.html HTTP/1.0" 200 2881
```

Web 日志记录是 Hadoop 大数据计算的典型数据源，因为它们代表每个用户的点击流式数据，从中可以轻松探究数据的多个方面。此外，它们往往也是非常大的动态半结构化数据集，非常适合在 Spark 和 MapReduce 中进行运算。要对该数据集进行初始计算，则先需要进行频率分析。例如，可以使用复合键将文本分解为两个每日时间序列，一个用于本地流量，另一个用于远程流量。

```
import re
from datetime import datetime

# 解析日志记录中的日期时间
dtfmt = "%d/%b/%Y:%H:%M:%S %z"

# 使用正则表达式解析日志记录
linre = re.compile(r'^(\w+) \- \- \[(.+)\] "(.+)" (\d+) ([\d\-]+)$')

def parse(line):
    # 使用正则表达式匹配日志记录
    match = linre.match(line)
    if match is not None:
        # 正则表达式含有分组,能提取日志源、时间戳、
        # 请求、状态码和响应字节大小
        parts = match.groups()

        # 解析日期时间,返回日志源、年和天
        date  = datetime.strptime(parts[1], dtfmt).timetuple()
        return (parts[0], date.tm_year, date.tm_yday)
```

将此函数用于 mapper 可以解析日志文件的每一行，或作为闭包传递给从文本文件中加载的 RDD 的 `map` 方法。`parse` 函数使用一个日期格式和一个正则表达式来解析行文本，然后发出由流量类型、年份和当天在当年中的位置组成的复合键。该键与计数器（例如数字 1）相关联，计数器可以被传递到求和 reducer 中以获取基于频率的时间序列。映射该数据集生成了以下数据：

```
('local', 1995, 120)  1
('local', 1995, 120)  1
('remote', 1995, 120)  1
('remote', 1995, 120)  1
```

用作**复杂**键的复合键支持在键空间的多个面上进行计算，例如网络流量的来源、年份和天，这是复合键最常见的用例。另一个常见用例是将特定键的信息传播到下游计算，例如依赖于 reduce 或每个键的聚合值的计算。通过将 reducer 的计算与键（特别是类似计数或浮点数的值）相关联，这些信息能与键一起被维护，以用于更复杂的计算。

MapReduce、Spark 的 Java 以及 Scala API 都需要使用强类型的键和值。就 Hadoop 而言，这意味着复合键和结构化的值需要被定义为实现 `Writable` 接口的类，并且键必须实现 `WritableComparable` 接口。这些工具使 Java 和 Scala 开发人员能够使用轻量级且可扩展的数据结构序列化功能，从而最大限度上减少了网络流量并支持 shuffle 和 sort 操作。然而，Python 开发人员需要自己将元组和 Python 原始类型序列化成字符串，还得将它们转换回来。要想序列化嵌套数据结构，可以使用 `json` 模块。在处理更复杂的作业时，二进制序列化格式（如 Protocol Buffers、Avro 或 Parquet）可通过最小化网络流量来缩短处理时间。

复合数据序列化

使用复合键（和复杂的值）的最后一步是理解复合数据的**序列化**和**反序列化**。**序列化**是指将内存中的对象转换成字节流，使其可以被写入磁盘或通过网络传输（**反序列化**是指相反的过程）。序列化过程必不可少，特别是在 MapReduce 中，因为键和值在 map 阶段和 reduce 阶段之间被写入磁盘（通常作为字符串写入）。然而，理解 Spark 中的序列化也非常重要——Spark 的中间作业要对数据进行预处理，供后续计算使用。

在 Spark 中，Python API 默认使用 `pickle` 模块进行序列化，这意味着你使用的任何数据结构都必须是可以 pickle 的。虽然 `pickle` 模块非常高效，但在 Spark 编程中，这个约束可能沦为一个陷阱（gotcha），特别是传递闭包（不依赖于全局值的函数，通常是匿名 lambda 表达式）时。使用 MapReduce Streaming 时，必须将键和值序列化为字符串，并以指定的字符分隔，默认情况下为制表符（`\t`）。新的问题来了：有没有更高效的办法，能将复合键（和值）序列化为字符串呢？

我们通常会简单地使用内置的 `str` 函数对不可变类型（例如元组）进行序列化，将该元组转换为可以轻松 pickle 或流式传输的字符串。然后问题转向反序列化——通过 Python 标准库中的 `ast`（抽象语法树）模块，使用 `literal_eval` 函数评估元组字符串得到 Python 元组类型，如下所示：

```
import ast

def map(key, val):
    # 解析复合键,它是一个元组
    key = ast.literal_eval(key)

    # 以字符串写新的键
    return (str(key), val)
```

随着键和值越来越复杂，你也得考虑使用其他序列化数据结构了，尤其是那种更紧凑的、能减少网络流量或能转换为字符串值以确保安全性的数据结构。例如，结构化数据的常见表示形式是 Base64 编码的 JSON 字符串，因为它很紧凑，仅使用 ASCII 字符，并且很容易用标准库进行序列化和反序列化，如下所示：

```
import json
import base64
```

```
def serialize(data):
    """
    返回数据(键或值)的Base64编码的JSON表示
    """
    return base64.b64encode(json.dumps(data))

def deserialize(data):
    """
    解码Base64编码的JSON数据
    """
    return json.loads(base64.b64decode(data))
```

但使用复杂的序列化表示时要小心，通常需要权衡序列化的计算复杂度与它所需的空间。许多类型的并行算法更适合使用元组字符串或制表符分隔格式，实现起来更快、更简单，如果能管理好键在计算中的传递过程更能事半功倍。在下一节中，我们将看到 MapReduce 和 Spark 中常见的基于键的计算模式。

5.1.2 键空间模式

可以使用键计算的概念管理数据集及其关系。然而，键也是计算的主要部分。因此，除了数据之外，键也必须被管理。本节将探讨影响**键空间**的几种模式，特别是爆炸（explode）、过滤、变换和恒等（identity）模式。这些常见模式通过键和值之间的关联关系来构造更大的模式并完成算法。

以下示例的主角是一个订单数据集，其中键是订单 ID、客户 ID 和时间戳，值是订单所购买产品的商品统一代码（universal product code，UPC）列表，如下所示：

```
1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865
1002, 0171488, 2014-12-11 03:05:03, 098668318865
1003, 1022739, 2015-01-03 13:01:54, 098668275427, 098668331789, 098668274321
```

1. 键空间变换

最常见的基于键的运算是输入键的域的变换，在 `map` 和 `reduce` 中均可进行。在 map 期间变换键空间会导致数据在聚合期间重新分区（划分），而在 reduce 期间变换键空间可用于重组输出（或后续计算的输入）。最常见的变换函数是直接赋值、复合、分割和键值换位。

直接赋值丢弃了输入的键（通常被完全忽略），从输入值或别的来源（例如随机的键）构造新键。思考一下从文本、CSV 或 JSON 加载原始或半结构化数据的情况。在这种情况下，输入键是行或文档 ID，通常因为某些特定于数据的值而被丢弃。

如上一节所述，**复合**及其相反运算**分割**管理复合键。复合构建复合键，或向复合键添加新的元素，能增加键关系的面；分割拆分复合键，而只使用其中一小部分。通常，值也能被复合和分割，复合键从拆分值接收新的数据（反之亦然），以确保没有数据丢失。此外，也可以通过复合或分割，丢弃不需要的数据或删除无关的信息。

键值换位交换键和值，是一种常见模式，特别是在链式的 MapReduce 作业或依赖于中间聚合（特别是 `groupby`）的 Spark 操作中。例如，为了通过值而不是键对数据集进行排序，必须先在 map 中将键和值换位，执行 `sortByKey` 或者利用 MapReduce 中的 shuffle 和 sort，然后在 reduce 或另一个 map 中重新换位。

来看一个按照每个订单中的产品数量和日期为订单排序的作业，这将使用之前学过的所有键空间变换方法：

```
# 将订单加载到一个RDD,解析CSV
orders = sc.textFile("orders.csv").map(split)

# 键分配:(orderid, customerid, date), products
orders = orders.map(lambda r: ((r[0], r[1], r[2]), r[3:]))

# 计算订单大小,并将键拆分为orderid和date
orders = orders.map(lambda (k, v): ((k[0], parse_date(k[2])), len(v)))

# 交换键和值,排序
orders = orders.map(lambda (k, v): ((v, k[1]), k[0]))

# 根据键将订单排序
orders = orders.sortByKey(ascending=False)

# 再次交换键和值,以便再次使用订单ID作为键
orders = orders.map(lambda (k,v ): (v, k))

# 根据订单大小和日期,获取前10个订单ID
print orders.take(10)
```

这个例子对于所需完成的任务可能有点冗长，但它确实演示了如下每种类型的变换。

(1) 如第 4 章讨论的，首先使用 `split` 方法从一个 CSV 文件中加载数据集。
(2) 此时的 `orders` 仅仅是一个列表集合，因此将值分解为 ID 和日期，将它们分配为键，将产品列表作为值，关联键与值。
(3) 下一步是获取产品列表的长度（订购的产品数量），并使用一个包装了 `datetime.strptime` 日期格式的闭包来解析日期。请注意，此方法拆分了复合键并删除了客户 ID，这其实没有必要。
(4) 为了将订单按大小排序，需要将订单大小的值和键换位，还要将日期从键中分割出来，以便也可以按日期排序。
(5) 执行排序后，键值重新换位，以便可以查看每个订单的大小和日期。

以下片段演示了第一条记录在经过 Spark 作业中的每个 map 时发生的变换：

```
0. "1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865"
1. [1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865]
2. ((1001, 1063457, 2014-09-16 12:23:33), [098668259830, 098668318865])
3. ((1001, datetime(2014, 9, 16, 12, 23, 33), 2)
4. ((2, datetime(2014, 9, 16, 12, 23, 33)), 1001)
5. (1001, (2, datetime(2014, 9, 16, 12, 23, 33)))
```

经过这一系列变换，客户端程序就可以按订单大小和日期获取前 10 个订单，并在分布式计算后将它们打印出来。

2. 爆炸mapper

爆炸 mapper 针对单个输入键生成多个中间键值对。一般来说，这是通过结合键移位（key shift）和将值拆分为多个部分来实现的。正如第 2 章中的单词计数示例，它根据空格拆分

行，将输入 mapper 中的单个行号 / 行对输出为几个新的键值对，即单词 /1。通过将值按组成部分划分并且重新将键分配给它们，爆炸 mapper 还可以生成许多中间对。

在此示例中，可以按订单展开产品列表，得到订单 / 产品对，如下面的代码所示：

```
def order_pairs(key, products):
    # 返回订单id/产品对的列表
    pairs = []

    for product in products:
        pairs.append((key[0], product))

    return pairs

orders = orders.flatMap(order_pairs)
```

将这个 mapper 应用到输入数据集，会产生如下输出：

```
1001, 098668259830
1001, 098668318865
1002, 098668318865
1003, 098668275427
1003, 098668331789
1003, 098668274321
```

注意在 RDD 上使用的 `flatMap` 操作，这是专为爆炸 map 设计的。它的操作与常规 map 相似，但该函数产生一个序列而不是单个项，然后该序列被链接成单个集合（而不是列表的 RDD）。在 MapReduce 和 Hadoop Streaming 中不存在这样的限制，一个 `map` 函数可以发射任意数量的对（或者根本不发送）。

3. 过滤器mapper

尽管本章后面会更详细地讨论过滤（特别是统计抽样），但鉴于它与键的操作相关，此处也得先说两句。过滤通常对限制 reduce 阶段执行的计算量至关重要，特别是在大数据环境中。它还可将同一数据流的计算划分为两条路径，这是专为超大数据集设计的一种大型算法，是面向数据的分支方法。想想看在只选择了 2014 年订单的情况下，如何扩展订单示例（使用 Spark）：

```
from functools import partial

def year_filter(item, year=None):
    key, val = item
    if parse_date(key[2]).year == year:
        return True
    return False

orders = orders.filter(partial(year_filter, year=2014))
```

Spark 提供了一个过滤操作，它接受一个函数并转换 RDD，只保留函数返回 `True` 的元素。此示例展示了闭包的高级用法以及可以获取任何年份的通用过滤器函数。`partial` 函数创建一个闭包，其 `year_filter` 的参数 `year` 始终为 2014，支持更强大的功能。MapReduce 代码类似，但需要更多的逻辑：

```
def YearFilterMapper(Mapper):

    def __init__(self, year, **kwargs):
        super(YearFilterMapper, self).__init__(**kwargs)
        self.year = year

    def map(self):
        for key, value in self:
            if parse_date(key[2]).year == self.year:
                self.emit(key, value)

if __name__ == "__main__":
    mapper = YearFilterMapper(2014)
    mapper.map()
```

mapper 什么也不发射也完全没问题。因此，用于过滤器 mapper 的逻辑是仅在满足条件时才发射。通过使用基于类的 `Mapper`，并用想要过滤的年份简单地实例化类，可以获得与 `partial` 方法相同的灵活性。更高级的 Spark 和 MapReduce 应用程序可能从运行作业的命令上接收年份作为输入。

最后一个订单记录（订单 1003）被删除了，因为它是 2015 年的订单，过滤产生的其他数据与输入相同：

```
1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865
1002, 0171488, 2014-12-11 03:05:03, 098668318865
```

4. 恒等模式

MapReduce 中的最后一个常用键空间模式（一般不用于 Spark 中）是恒等（identity）函数。它只是一个传递，能使恒等 mapper 或者恒等 reducer 返回与输入相同的值（就好像在**恒等函数** `f(x) = x` 中一样）。恒等 mapper 通常用于在数据流中执行多个 reduce。当在 MapReduce 中使用恒等 reducer 时，该作业等同于在键空间上进行排序。恒等 mapper 和恒等 reducer 的简单实现如下所示：

```
class IdentityMapper(Mapper):

    def map(self):
        for key, value in self:
            self.emit(key, value)

class IdentityReducer(Reducer):

    def reduce(self):
        for key, values in self:
            for value in values:
                self.emit(key, value)
```

因为 MapReduce 使用了最优的 shuffle 和 sort，因而恒等 reducer 通常更常见一些。不过恒等 mapper 也非常重要，特别是在链式 MapReduce 作业中，一个 reducer 的输出必须立即被第二个 reducer 再次 reduce。事实上，正是因为 MapReduce 的操作是分阶段的，所以才需要恒等 reducer。在 Spark 中，因为 RDD 被延迟评估，所以不需要恒等闭包。

5.1.3 pair与stripe

数据科学家习惯用表示为向量、矩阵或数据框的数据。线性代数计算往往针对单核机器进行了优化，而机器学习中的算法使用低级数据结构（如 numpy 库中的多维数组）来实现。这些结构虽然紧凑，但因为数据的量级实在太大，所以无法在大数据环境中使用。相反，矩阵通常有两种表示方式：pair 和 stripe。pair 和 stripe 都是基于键的计算。

为了理解这一点，试着为基于文本的语料库建立单词共现矩阵（和单词计数一样，这是演示 pair 和 stripe 的典型问题。[1]）使用单词共现创建的语言的统计模型可用于机器翻译、句子生成等大量应用。

如图 5-2 所示的单词共现矩阵是大小为 N×N 的矩阵，其中 N 是语料库的词汇量（单词的种数）。每个单元 $W_{i,j}$ 包含词 w_i 和词 w_j 同时出现在句子、段落、文档或其他固定长度窗口中的次数。这个矩阵很稀疏，特别是采用了积极的停用词过滤之后，因为大多数单词通常仅与非常少的其他词共现。

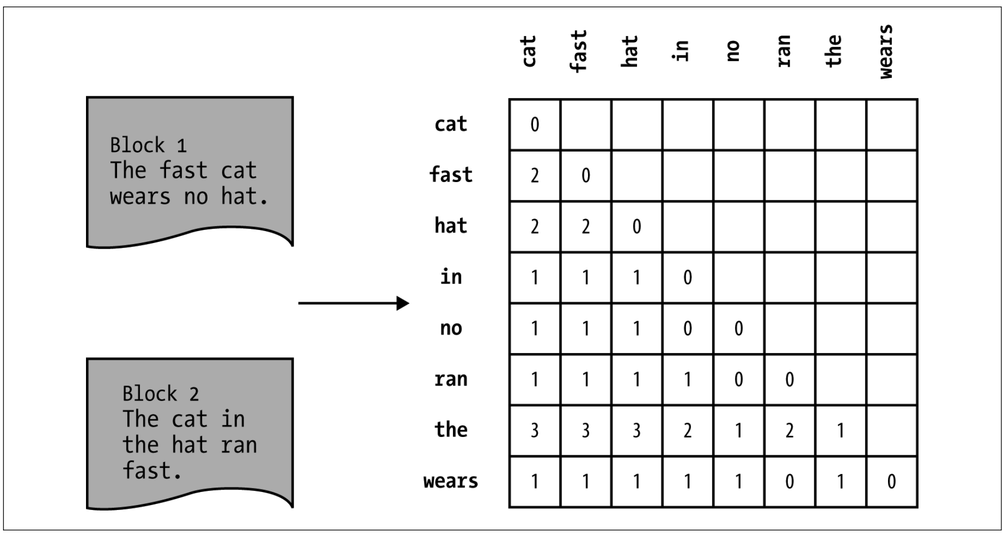


图 5-2：演示同一文本块（例如句子）中词条同时出现频率的单词共现矩阵

pair 方法将矩阵中的每个单元映射到特定值，其中词对是复合键 i,j。因此，reducer 对每个单元的值进行处理，以产生最终的单元挨单元的矩阵。这是一种合理的方法，它产生的输出中的每个 $W_{i,j}$ 被单独计算并存储。使用求和 reducer，mapper 如下所示（有关使用 NLTK 进行文本处理和分词的更多信息，请参见第 3 章）：

```
from itertools import combinations

class WordPairsMapper(Mapper):

    def map(self):
```

注 1：Jimmy Lin, Chris Dyer, *Data-Intensive Text Processing with MapReduce* (http://bit.ly/1PcgEmB).

```
        for docid, document in self:
            tokens = list(self.tokenize(document))
            for pair in combinations(sorted(tokens), 2):
                self.emit(pair, 1)
```

这个方法的重中之重是使用内置的 `sorted` 函数对令牌进行**字典排序**。在对称矩阵（$W_{i,j}$ 等于 $W_{j,i}$ 的矩阵）中，必须为词对排序，否则键 `(b,a)` 和 `(a,b)` 不会 reduce 在一起。请注意，`itertools` 库中的 `combinations` 函数保持其输入列表的排序。输入如下所示：

```
"See Spot run, run Spot, run!"
```

单词共现矩阵的词对聚合结果如下所示：

```
(run, run), 6
(run, see), 3
(run, spot), 6
(see, run), 3
(see, spot), 2
(spot, run), 6
(spot, see), 2
(spot, spot), 1
```

虽然 pair 方法易于理解和实现，但是它产生了许多中间对。这些中间对必须在网络上传输，这一过程在 MapReduce 的 shuffle 阶段和 sort 阶段，以及 `groupByKey` 操作将值 shuffle 到 RDD 各分区的期间均有发生。此外，pair 方法不太适用于需要整行（或列）数据的计算。

stripe 方法最初被设想为一种减少中间对的数量和网络通信的优化手段，从而让作业运行得更快。不过它很快也成为一种重要的工具，应用于许多需要针对一个元素执行快速计算（例如相对频率或其他统计运算）的算法中。stripe 方法没有使用词对，而是在 mapper 中为每个条目构造关联数组（Python 字典），并作为值发射：

```
from collections import Counter

class WordStripeMapper(Mapper):

    def map(self):
        for docid, document in self:
            tokens = list(self.tokenize(document))

            for i, term in enumerate(tokens):
                # 为每个条目创建新的stripe
                stripe = Counter()

                for j, token in enumerate(tokens):
                    # 不计算该条目与本身的共现
                    if i != j:
                        stripe[token] += 1

                # 发射条目和stripe
                self.emit(term, stripe)

class StripeSumReducer(Reducer):
```

```
        def reduce(self):
            for key, values in self:
                stripe = Counter()

                # 将所有计数器相加
                for value in values:
                    for token, count in value.iteritems():
                        # 为每一个令牌分别累加stripe
                        stripe[token] += count

                self.emit(key, stripe)
```

stripe 的 mapper 和 reducer 有点复杂。在 mapper 中需要对所有令牌进行两层嵌套循环，并且必须确保该条目不对其自身计数。内置的 `enumerate` 函数允许我们在两层循环中跟踪条目的索引，跳过相同的索引而不是条目（如果条目在文本中重复出现，则该条目实际上可能共现）。`collections` 库中的 `Counter` 是一个有用的数据结构，它本质上是字典，默认值是 `int`。然后，reducer 需要对字典中的每个元素进行求和，计算 mapper 中所有计数器的总数。虽然输入相同，但现在的输出更紧凑：

```
run, ((run, 6), (see, 3), (spot, 6))
see, ((run, 3), (spot, 2))
spot, ((run, 6), (see, 2), (spot, 1))
```

stripe 方法不仅在其表示上更紧凑，而且也生成更少、更简单的中间键，从而优化了数据的 sort、shuffle 等方面。然而，stripe 对象更庞大，在处理时间和序列化方面的开销都更大，特别是当 stripe 非常大时。stripe 的大小有上限，尤其当共现矩阵非常密集时，可能需要大量的内存来记录一个条目的数据。

介绍完 pair 和 stripe，键计算的讨论也就结束了。正如你看到的，大多数大数据计算都靠基于键的计算来提供和维护数据集之间的关系，以确保数据能合理分布在不同的 mapper 和 reducer 上。在 Spark 和 MapReduce 上执行大规模计算需要我们换个角度思考标准计算的传统方法，因为数据的规模实在太大。下一节将基于这种思维转变，提出具体的设计模式，这些模式通常用于分解以实现最后一英里计算。

5.2 设计模式

设计模式是软件设计中的一个特殊术语，是指针对特定编程挑战的通用、可重用解决方案。设计模式通常不限定语言，不仅指模式的实现细节，更指设计或构造策略。最常见的软件设计模式可能是在 Web 开发中非常流行的模型 – 视图 – 控制器（model-view-controller，MVC）模式，可以使用 Ruby、Java 等各种语言实现。

与之类似，我们也可以探索函数式设计模式，用于解决 MapReduce 和 Spark 中的并行计算问题。这些模式展示了可用于更复杂或特定于领域的角色的通用策略和原则。事实上，我们已经在用于计算单词共现的 pair 和 stripe 模式中看过一个例子。pair 和 stripe 都可以应用于更一般的计算。

在《MapReduce 设计模式》中，Donald Miner 和 Adam Shook 探索了 MapReduce 作业的 23

种常见设计模式。他们将模式大致分为如下几类。

概要
: 根据聚合、分组、统计度量、索引或数据的其他高级视图，提供大型数据集的摘要视图。

过滤
: 基于一组固定的条件创建数据的子集或样本，并且不以任何方式修改原始数据。

数据组织
: 将记录重组为有意义的模式，不一定使用分组。此任务适合作为后续计算的第一步。

连接
: 从不同来源将相关数据收集到一个统一的整体。

元模式
: 为复杂或优化过的计算实现作业链和作业合并。这类模式是与其他模式相关联的模式。

输入和输出
: 使用数据处理模式对输入源的数据进行转换，以输出到一个不同的输出源。输入源和输出源可以位于 HDFS，也可以是其他数据源。

本节将详细探讨 MapReduce 和 Spark 的概要和过滤技术，并概述 MapReduce 数据流和作业链的构造。我们将用几个具体应用来揭示这些模式背后的秘密，但有一点非常显而易见：这些技术通常将输入数据分解或转换成用于最后一英里计算的较小数据源。

5.2.1 概要

概要尝试用尽可能简单的方法描述尽可能多的关于数据集的信息。我们习惯于阅读内容提要（executive summary），它突出冗长文件的主要内容，而不涉及细节。与之类似，描述性统计数据试图通过测量观测数据的集中趋势（平均值、中值）、分散情况（标准差）、分布形状（偏度）或变量之间的依赖关系（相关性）来概括观察数据之间的关系。

基于键的计算对数据进行分组（另一种形式的概要），聚合某个通常能描述键的值（这可能有价值），然后概要计算就进入了下一步。基于键的计算可以是简单的分析，例如计算最不准点的机场或航空公司；也可以是稍复杂些的分析，例如推断天气、距离或其他因素对机场或航空公司的影响。在很多情况下，概要只是更大的概括和预测的第一步，例如作为语言模型的单词共现的计算，或描述概率分布的频率分析。

从原理上讲，MapReduce 和 Spark 是应用一系列的概要，将具体的数据形式（每个单独的记录）转换为更一般的形式。大体来说，我们最熟悉的概要具备以下操作特征：

- 聚合（集合到单个值，如平均值、总和或最大值）
- 索引（将值映射到值集合的函数映射）
- 分组（将集合选择或划分成多个集合）

在本节中，我们将探索聚合和索引的模式（通过之前讨论的基于键的技术可以轻松实现分组）。具体来说，我们将探索数据集的并行统计描述，例如 Pandas 中的 `describe` 命令以

及如何实现聚合；然后，将探索两种索引技术：倒排索引和通过词频 – 逆文档频率（term frequency-inverse document frequency，TF-IDF）的文档概要。

1. 聚合

MapReduce 和 Spark 上下文中的聚合函数拥有两个输入值并生成一个输出值，它也满足**交换律**和**结合律**，因此可以并行计算。回顾一下，交换律表明顺序对二元运算没有影响，例如对给定的运算 ✤ 来说，`a ✤ b = b ✤ a`；结合律表示无论输入如何分组，计算都是相同的，即（`a ✤ b`）✤ `c = a ✤`（`b ✤ c`）。加法和乘法是满足交换律和结合律的，但减法和除法不是。

聚合一般是对一个集合应用操作以创建较小集合（聚集在一起），而 reduce 通常被认为是将集合 reduce 为单个值的操作。聚合也可以被认为是一系列更小的 reduce 组成的应用程序。在这种情况下，为什么结合性和交换性是实现并行不可或缺的条件也就显而易见了：给定一个 reduce `a ✤ b ✤ c ✤ d`，由于网络或其他物理约束导致 shuffle 的结果不确定，这意味着顺序不能有影响。结合性允许一个进程计算 `a ✤ b`，另一个进程计算 `c ✤ d`，其中一个进程发送它们的结果以执行最终的 ✤ 操作。

想想标准数据集描述量：平均值、中值、众值、最小值、最大值、总和。其中，**总和**、**最小值**和**最大值**都容易实现，因为它们都满足结合律和交换律；但**平均值**、**中值**和**众值**不是。对中值和众值来说，通常需要先进行某种排序，而由于涉及除法，分组计算平均值会产生精度损失。虽然这些计算有并行的近似算法，但更重要的是意识到，在执行这些类型的分析时应格外小心。我们将使用一个 MapReduce 作业来描述整个数据集，而不是单独讨论这些计算。

2. 统计概要

现在，可以通过两个关键概念来分析处理数据集。首先，使用键作为关系来定义有意义的数据子集；其次，使用多种方法，包括作业链、键空间管理或 pair、stripe 等机制，同时实现多个计算。通过将实例按键分组并描述属性（类似于 Pandas 中的 `describe` 命令），可以简化大型数据集并对其执行概要分析。

我们已经见过了按键计算均值和计数的例子，这些类型的分析通常最先运行，从而对较大数据集有初步了解。特别是对高速变化的大数据（以惊人速度变化的数据）来说，运行定期描述性作业可以让你了解已经发生的变化以及它们是如何变化的。我们将在一个批次中一次性运行所有的 6 个作业，包括计算总数、总和、平均值、标准差和范围（最小值和最大值），而不是为每个描述性指标单独实现一个 MapReduce 作业（代价高昂）。

这时的基本策略是，为所有按键进行的计算映射一个计数器值的集合。然后，reducer 将每个操作分别应用于值集合中的每个项目，使用每个项目来计算最终输出（例如平均值取决于总数和总和）。这样的 mapper 的基本结构如下所示：

```
class StatsMapper(Mapper):

    def map(self):
        for key, value in self:
            try:
                value = float(value)
                self.emit(key, (1, value, value ** 2))
```

```
            except ValueError:
                # 无法解析,忽略
                pass
```

在这种情况下，直接进行 reduce 的三种操作是计算总数、总和和平方和。因此，该 mapper 针对每个键发射一次，1 用于计数，值用于求和，值的平方用于平方和。reducer 使用总数及总和来计算平均值，使用值计算范围，使用总数、总和以及平方和计算标准差，如下所示：

```
from ast import literal_eval as make_tuple

class StatsReducer(Reducer):

    def reduce(self):
        for key, values in self:
            # 解析mapper发送过来的值
            values = make_tuple(values)

            count   = 0
            delay   = 0.0
            square  = 0.0
            minimum = None
            maximum = None

            for value in values:
                count  += value[0]
                delay  += value[1]
                square += value[2]
                if minimum is None or value[1] < minimum:
                    minimum = value[1]

                if maximum is None or value[1] > maximum:
                    maximum = value[1]

            mean   = delay / float(count)
            stddev = math.sqrt((square-(delay**2)/count)/count-1)

            self.emit(key, (count, mean, stddev, minimum, maximum))
```

这个作业举例说明了如何将复杂数据类型作为 MapReduce 的输出和中间值使用，这是迈向高级分析方法的第一步。reducer 使用 `ast.literal_eval` 反序列化机制来解析值元组，然后对数据值执行**单个**循环（你必须将所有值加载到内存中，例如作为列表，以进行多次遍历）来计算各种和、最小值和最大值。

MapReduce 中的 reducer 可以访问与单个键相关联的所有值的可迭代对象，而在 Spark 中，必须对这个计算稍作修改。Spark 中的 reduce 不能应用以集合作为输入的操作，因而你必须每一次将操作应用于输入中的一对元素。又因为第一次应用的结果是第二次应用的第一个输入，所以操作必须满足结合律和交换律。例如，对于给定输入 `[5, 2, 7]`，你不能简单地将 `sum` 应用于集合，而要这样使用 `add`：`add(add(5, 2), 7)`。因此，必须为 mapper 输出的值添加最小值和最大值计数器，以分别记录 reduce 过程中的最小值和最大值，如下所示：

```
def counters(item):
    """
    将键值对解析为键和概要计数器
    计数器格式：(count, total, square, minimum, maximum)
    """
    key, value = item # 分解item元组
    try:
        value = float(value)
        self.emit(key, (1, value, value ** 2, value, value))
    except ValueError:
        # 无法解析,忽略
        pass

def aggregation(first, second):
    """
    对两个(key, counter)执行概要聚合
    """
    count1, total1, squares1, min1, max1 = first
    count2, total2, squares2, min2, max2 = second
    minimum = min((min1, min2))
    maximum = max((max1, max2))
    count   = count1 + count2
    total   = total1 + total2
    squares = squares1 + squares2

    return (count, total, squares, minimum, maximum)

def summary(aggregate):
    """
    根据聚合结果,计算概要统计
    """
    (key, (count, total, square, minimum, maximum)) = aggregate

    mean   = total / float(count)
    stddev = math.sqrt((square-(total**2)/count)/count-1)

    return (key, (count, mean, stddev, minimum, maximum))


def main(sc):
    """
    Spark应用程序的主要分析过程
    """

    # 给定一个键值对数据集,映射到counters
    dataset = dataset.map(counters)

    # 根据键执行概要聚合
    dataset = dataset.reduceByKey(aggregation)
    dataset = dataset.map(summary)

    # 将结果写入磁盘
    dataset.saveAsTextFile("dataset-summary")
```

reduceByKey 函数的规则使得 Spark 作业中的数据流不太一样。我们无法通过迭代跟踪最小

值和最大值，而只能在计算结果中标注出**最后看到**的最小值和最大值，并在继续 reduce 时传播它们。因此，我们不能在聚合期间简单地执行最终计算，而是需要另一个 map 在聚合后的 RDD（小得多）上完成概要计算。

这个 `describe` 示例提供了一种有用的模式，可以同时计算多个特征并将它们作为向量返回。这种模式经常被重用，在机器学习上下文中尤其如此，因为可能需要多个过程生成训练所需的实例（例如二次计算、归一化、插补、连接或更多具体的机器学习任务）。理解 MapReduce 聚合和 Spark 聚合之间的差异，对跟踪错误以及在 MapReduce 和 Spark 之间进行代码移植大有帮助。

5.2.2 索引

与基于聚合的概要技术不同，**索引**采用多对多的方法。聚合将多个记录收集到单个记录中，而索引将多个记录与一个或多个索引相关联。在数据库中，索引是用于快速查找的专用数据结构，通常是二叉树（binary-tree，B-Tree）。在 Hadoop/Spark 中，索引也能发挥类似的功能，但是它们不会被维护和更新，而通常会成为需要快速查找的下游计算的第一步。

文本索引在 Hadoop 算法“万神殿”中地位特殊，这是由于 Hadoop 最初被用于创建搜索应用程序。当仅处理一小部分文档时，它可以像 `grep` 一样扫描文档来查找搜索项。然而，随着文档和查询的数量增加，再使用这种方法就不合适了。在本节中，我们将看到两种类型的基于文本的索引：常见的倒排索引以及词频 – 逆文档频率（TF-IDF）。TF-IDF 是与索引相关联的数字统计量，通常用于机器学习。

1. 倒排索引

倒排索引是从索引项到文档集合中的位置的映射（与从文档到索引项映射的前向索引相反）。在全文搜索中，索引项是搜索项，通常是去除了停用词（例如在搜索中无意义的常见词）的词或数字。大多数搜索引擎还采用了某种词干提取（stemming）和词形还原（lemmatization）：具有相同含义的多个词被分类到单个词类（例如“running”“ran”“runs”由单个词语“run”索引）。

最常见的倒排索引用例是搜索：它让搜索算法能快速检索出要排列和返回的文档子集，而免于扫描每个文档。例如，要想查询“running bear”，可以用索引查找包含搜索项“running”和包含搜索项“bear”的文档的交集。然后采用简单的排名系统来返回搜索项紧挨着，而不是相距很远的文档（现代搜索排名系统显然比这要复杂得多）。

然而，搜索示例可以被一般化到机器学习上下文。索引项不一定是文本，它可以是一条更长的记录的任意部分。此外，使用索引来简化或加快下游计算（如排名）的任务也很常见。根据索引的创建方式，可以在性能和准确率之间进行权衡；或者在给定随机索引的情况下，在精确率和召回率之间进行权衡。

来考虑某个预处理后的文本，其中文档 ID 和行号作为键，行文本作为值。这种预处理方式可以用于所有用户驱动的文本，如留言板或评论；但在这里，我们将它用于莎士比亚戏剧全集。具体来说，我们要创建一个人物关联索引。因此，不能将人物映射到已有的行，而是要针对人物和起始行进行概要分析，以便看出人物的出场顺序。语料库中的每一行表

示如下：

```
hamlet@15261    HAMLET    O, that this too too solid flesh would melt
hamlet@15261              Thaw and resolve itself into a dew!
```

行的第一部分是 `title@lineno`（戏剧名 @ 行号）标识符，然后是一个 TAB 字符（`\t`）、人物的名字、第二个 TAB 字符和剧本中的一行文本。如果相同的人物连续说了多行，则用两个 TAB 字符将标识符与文本分开。为了创建一个人物的倒排索引，我们将使用一个恒等 reducer 和下面的 mapper（注意 Spark 中很容易实现相同的算法）：

```
class CharacterIndexMapper(Mapper):

    def map(self):
        for row in self:
            row = row.split("\t")            # 使用制表符拆分
            if not len(row) >= 3: continue    # 确保数据格式

            if row[1] != "":
                # 如果存在人物,发射名字和docid/lineno
                self.emit(row[1], row[0])
```

这个莎士比亚人物索引示例说明了索引的几个关键点。首先，索引项可以是任意的（这里是人物名称）；其次，这种算法虽然非常简单，但它高度依赖于输入数据的结构（例如，我们知道要搜索 TAB 分片以找到人物名称）；最后，这个示例还使用了我们之前看到的一些 map 和 reduce 模式——恒等 reducer。继续发展的话，这个数据结构可以创建人物之间的对话图，更可以用于分析人物所在的人际圈或人物相似性。最关键的一点是，倒排索引通常是下游计算的第一步。

人物索引作业的输出是人物名称的列表，每个人名对应该人物每次开始说话的行的列表，这可以当作查找表或作为其他类型分析的输入使用。

2. TF-IDF

词频 – 逆文档频率（TF-IDF）可能是目前最常用的基于文本的概要形式，也是基于文本的机器学习中最常用的文档特征。TF-IDF 是一种指标，定义词条（单词）和作为较大语料库一部分的文档之间的关系。具体来说，它给出该词在其他文档中的相对频率，从而试着定义该词对于特定文档的重要性。

词频 $tf_{i,j}$ 是给定词条 i 在文档 j 中出现的次数，通常用于衡量该词与该文档的相关性。以一份关于美国政治的文件为例：一方面，我们可能会说像“民主”（democracy）或“选举”（election）这样的词语比“鲁米那”（luminal）这样的词语出现得更频繁，因此它们与文档的整体论述更相关；另一方面，词频本身将过度强调常见的词语，如“说”（speaking）——在给定组合语料库中，该词会出现于科学和政治类文档中。因此，词条 i 的文档频率 df_i，即该词条在多少文档中出现过，用于弥补词频的片面性。也就是说，包含词条的文档数与文档总数 `N` 的比例的倒数的对数与词频相乘。TF-IDF 分数高，则给定词经常在目标文档中出现，但不常在语料库的其他位置出现。文档 j 中的词条 i 的 TF-IDF 如下

所示：

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

这种方法最初用于文档的主题建模，这是一种试图将主题相同的文档相互关联的聚类形式。不难看出，若文档共享高 TF-IDF 值的词，它们则可能彼此相关，因为这些词条不常出现在语料库的其余部分。出于类似的原因，TF-IDF 现在被广泛应用于其他机器学习任务，包括分类、自动问答，甚至非结构化数据的社交网络或 Web 分析中。

在索引中加入该算法，原因和加入简单的倒排索引类似：它创建了通常用于下游计算和机器学习的数据结构。此外，这个更复杂的示例突出了其他几节只简单涉及的内容：使用作业链实现单个算法。考虑到这一点，让我们来看看 TF-IDF 的 MapReduce 实现。

我们的策略是使用键空间模式在三个作业中传播所需的数据：第一个作业使用简单的单词计数来计算每个文档的词频，该单词计数还维护该词的文档 ID；第二个作业计算该词一共出现在多少文档中；最后一个作业使用前两个作业传播到最后的信息计算 TF-IDF。第一个作业如下所示：

```
class TermFrequencyMapper(Mapper):

    def __init__(self, *args, **kwargs):
        """
        初始化分词器和停用词
        """
        super(TermFrequencyMapper, self).__init__(*args, **kwargs)

        self.stopwords = set()
        self.tokenizer = re.compile(r'\W+')

        # 从文本文件读取停用词
        with open('stopwords.txt') as stopwords:
            for line in stopwords:
                self.stopwords.add(line.strip())

    def tokenize(self, text):
        """
        对一行文本进行分词和规范化(只产生非数字、标点和空字符串的非停用词)
        """
        for word in re.split(self.tokenizer, text):
            if word and word not in self.stopwords and word.isalpha():
                yield word

    def map(self):
        for docid, line in self:
            # 对每一行分词,并发射每个(word, docid)
            for word in self.tokenize(line):
                self.emit((word, docid), 1)

class SumReducer(Reducer):

    def reduce(self):
```

```
        for key, values in self:
            total = sum(int(count[1]) for count in values)
            self.emit(key, total)
```

为了计算文档中的词，不能简单地使用空格分割行，而是要使用正则表达式对文本进行分词——这可能会因为索引需要而变得更复杂。我们还从 stopwords.txt 文件中读取了停用词列表，该文件需要包含在作业中。因此，我们的分词方法简单地使用正则表达式进行拆分，并过滤掉停用词、数字和标点符号。更高级的分词器也可以提取词干，或者实现归一化（例如全部变为小写）。第一个作业发射 `(term, docid)` 为键、频率为值的元组。

第二个作业由一个 mapper 和一个 reducer 组成，如下所示：

```
class DocumentTermsMapper(Mapper):

    def map(self):
        for line in self:
            key, tf = line.split(self.sep)  # 将每一行拆分成键值对
            word, docid = make_tuple(key)   # 解析元组字符串
            self.emit(word, (docid, tf, 1)) # 发射词和带计数器的数据

class DocumentTermsReducer(Reducer):

    def reduce(self):
        for word, values in self:
            # 将values加载到内存,进行多次处理和解析
            values = [make_tuple(value) for value in values]

            # 第一次处理:计算词条的文档频率
            terms = sum(int(item[2]) for item in values)

            # 第二次处理:为与docid关联的每一个word发射一个值
            for docid, tf, num in values:
                self.emit((word, docid), (int(tf), terms))
```

此作业的 mapper 又是一个计数 mapper，用于求词条的文档频率的和；它还改变了键空间，维护针对该文档的词频并将文档 ID 添加到值中。这样，我们可以按词 reduce，其中每个值都对应一个文档。因此，reducer 需要遍历数据两次：一次求和，另一次执行每个文档的键空间更改。为了做到这一点，必须将元组 `(docid, tf, count)` 缓存在内存中，使用列表推导从生成器加载数据。如果许多文档都包含该词（如“the”这样的高频词），这个计算也许不能在内存中进行。也正因如此，停用词列表对 TF-IDF 的计算才如此重要。其他解决方法包括：将中间数据临时存储到磁盘；再实现一个中间 MapReduce 作业，一个作业用于求词条的文档频率的和，另一个用于改变键空间：

```
class TFIDFMapper(Mapper):

    def __init__(self, *args, **kwargs):
        self.N = kwargs.pop("documents")
        super(TFIDFMapper, self).__init__(*args, **kwargs)

    def map(self):
        for line in self:
            key, val = map(make_tuple, line.split(self.sep))
```

```
            tf, n = (int(x) for x in val)
        if n > 0:
            idf = math.log(self.N/n)
            self.emit(key, idf*tf)
```

最后一个作业是一个只有 map 的作业，因为我们已经有了计算用的键——由上一个 reducer 发射的 (word, docid) 对。使用恒等 reducer 就完全能够搞定。我们简单地将行解析为 int 的元组，并且只要频率大于零，就计算 TF-IDF。请注意，需要一条额外的信息——语料库中的文档数量，它在这个过程中没有参与计算。

这个任务虽然看似很复杂，但是将执行过程设想为数据流就能好很多：随着计算结果片段的产生，它们被添加到数据流中。键 / 值选择由计算中的下一步骤激发。而且最重要的是，这个计算仅仅遍历了一次原始输入，所以它支持作业的线性依赖。TF-IDF 计算的 Spark 实现也需要这种数据流思维模式，如下所示：

```
def tokenize(document, stopwords=None):
    """
    分词并返回(docid, word)和一个计数
    """

    def line_tokenizer(lines):
        """
        逐行分词的内部生成器
        """
        for line in lines:
            for word in re.split(tokenizer, line):
                if word and word not in stopwords.value and word.isalpha():
                    yield word

    docid, lines = document
    return [
        ((docid, word), 1) for word in line_tokenizer(lines)
    ]

def term_frequency(v1, v2):
    """
    拆分复杂的值,计算词频
    """
    docid, tf, count1 = v1
    _docid, _tf, count2 = v2
    return (docid, tf, count1 + count2)

def tfidf(args):
    """
    给定((word, docid), (tf, n))参数,计算TF-IDF
    请注意,必须提前定义N_DOCS,它是语料库中的文档数(n是word的文档频率)
    """
    (key, (tf, n)) = args
    if n > 0:
        idf = math.log(N_DOCS/n)
        return (key, idf*tf)

def main(sc):
```

```
"""
Spark应用程序的主要分析过程
"""

# 从数据集加载停用词
with open('stopwords.txt', 'r') as words:
    stopwords = frozenset([
        word.strip() for word in words.read().split("\n")
    ])

# 将停用词广播到集群
stopwords = sc.broadcast(stopwords)

# 第一阶段：分词并计算文档频率
# 请注意：假设有一个包含(docid, text)对的语料库
docfreq = corpus.flatMap(partial(tokenize, stopwords=stopwords))
docfreq = docfreq.reduceByKey(add)

# 第二阶段：计算词频,然后执行键空间更改
trmfreq = docfreq.map(lambda (key, tf): (key[1], (key[0], tf, 1)))
trmfreq = trmfreq.reduceByKey(term_frequency)
trmfreq = trmfreq.map(
    lambda (word, (docid, tf, n)): ((word, docid), (tf, n))
)

# 第三阶段:为每个(word,document)对计算TF-IDF
tfidfs = trmfreq.map(tfidf)
```

Spark 作业同样从磁盘加载停用词，然后将其广播到集群的剩余部分。然后，就可以对默认参数为停用词广播值的 `tokenize` 偏函数应用 `flatMap` 操作。这里之所以使用 `flatMap`，是因为 `tokenize` 函数将为文档中的每一行生成一个令牌计数列表（需要使用内部的 `line_tokenizer` 函数）。最后，将 Spark 实现的 `term_frequency` 和 `tfidif` 函数映射到每个文档。请注意，因为 `reduceByKey` 被调用了两次，并且需要在 `tfidfs` RDD 上应用某个最后的动作，所以此 Spark 作业同样具有三个数据流，和 MapReduce 作业一样。

5.2.3 过滤

过滤是粗粒度地减少下游计算数据的主要方法之一。与聚合通过宏观概览分组来缩小输入空间不同，过滤意在通过去除不需要的记录来缩小计算空间。在键空间一节中，我们探讨了应用于 mapper 的过滤。事实上，因为 mapper 非常适合执行过滤，所以许多过滤任务常使用只有 map 的作业（不需要 reducer）。这可以视为通过谓词或选择进行过滤，类似于 SQL 语句中的 `where` 子句。

另外一些过滤任务使用 reducer，收集具有代表性的数据集或根据值进行过滤。这种过滤包括查找最大的 `n` 个值或最小的 `n` 个值、去重或子选择。分析中非常常见的过滤任务是抽样：创建一个较小的、具有代表性的数据集，该数据集相对于较大的数据集分布良好（取决于你期望实现的分布类型）。开发中使用面向数据的子样本验证机器学习算法（例如交叉验证）或者进行其他统计计算（例如幂运算）。

我们通常可以将过滤实现为一个函数，它接受一条记录作为输入。如果评估返回 true，则

发射记录，否则丢弃记录。本节将探讨无序的最大 / 最小 *n* 个元素、抽样技术，以及经布隆过滤器提高性能后的高级过滤。

1. top n记录

top n 记录（以及相反的 bottom n 记录）方法是一个基数比较过滤器，它需要一个 mapper 和一个 reducer。其基本原理是让每个 mapper 产生其 top n 个项目，然后 reducer 将从 mapper 产生的项目中同样选择 top n 个项目。如果 n 相对较小（至少与数据集的其余部分相比），单个 reducer 应该能够轻松处理该计算，因为每个 mapper 最多产生 n 条记录：

```
import bisect

class TopNMapper(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(TopNMapper, self).__init__(*args, **kwargs)

    def map(self):
        items = []
        for value in self:
            # 维护有序的items列表
            bisect.insort(items, value)

        for item in items[-self.n:]:
            # 从mapper发射前n个值
            self.emit(None, item)

class TopNReducer(object):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(TopNReducer, self).__init__(*args, **kwargs)

    def reduce(self):
        items = []
        for _, values in self:
            for value in values:
                bisect.insort(items, value)

        for item in items[-self.n:]:
            # 从mapper发射前n个值
            self.emit(None, item)
```

这里的 mapper 和 reducer 都使用了 `bisect` 模块将值按升序插入列表。为了获得最大的 *n* 个值，使用了负索引的切片，从而选择有序列表中的最后 *n* 个值。要得到最小的 *n* 个值，可以简单地分片取出列表中的前 *n* 个值。使用 `None` 作为键可确保仅使用单个 reducer。请注意，Spark 拥有丰富的 RDD API，你可以使用 `top` 和 `takeOrdered` 动作，而不必自己实现。请注意，对于 Spark 和 MapReduce，为了能进行排序，需要记录是可比较的，这需要进行严格的解析；例如，在 Python 中，`'14'> 22` 为 `True`。

这种方法最大的好处是不需要对整个数据集进行完整的排序；相反，每个 mapper 对它们

自己的数据子集进行排序，而 reducer 仅看到 mapper 数量 n 倍的数据。这段代码可以通过几种方式进行优化，但主要优化与所使用的数据结构有关。下一节将研究如何在 bisect 模块上使用堆（heap）来实现类似的功能。

2. 简单随机抽样

简单随机抽样是数据集的子集，数据集的每条记录属于该子集的可能性相同。在这种情况下，评估函数不关心记录的内容或结构，而是利用某种随机数生成器来评估是否发射记录。但问题来了，如何确保每个元素被选中的可能性相同呢？

如果需要的样本不是必须大小为 n，而是包含百分之多少的记录就好，第一种方法是简单地使用随机数生成器来产生数字，并将其与期望的阈值大小进行比较。随机数生成器可用值的范围与阈值将共同决定大约发射百分之多少的记录。一般来说，随机数生成器返回的值在 0~1，因此与百分比的直接比较将产生预期的结果！例如，如果想要从数据集中采样 20%，可以写如下的 mapper：

```
import random

class PercentSampleMapper(Mapper):

    def __init__(self, *args, **kwargs):
        self.percentage = kwargs.pop("percentage")
        super(PercentSampleMapper, self).__init__(*args, **kwargs)

    def map(self):
        for _, val in self:
            if random.random() < self.percentage:
                self.emit(None, val)

if __name__ == '__main__':
    mapper = PercentSampleMapper(sys.stdin, percentage=0.20)
    mapper.map()
```

使用 percentage 关键字参数初始化 PercentSampleMapper，该参数从 __init__ 中的通用关键字参数中取出。此作业将返回原始数据集的 20% 左右，因为每条记录产生的随机数小于 0.2 的可能性相同，所以随机数小于 0.2 发生的可能性只大约为调用次数的 20%。如果这个作业运行时只有 mapper 而没有 reducer，许多小文件将被写入到磁盘，文件的数量与 mapper 的数量相同。使用一个恒等 reducer 将确保这些值都被收集到单个文件中。

然而，如果想要一个大小精确为 n 的样本呢？为了确保每种方法机会均等，必须进行 n 次随机选择，每次选择一个元素，而不进行替换，以确保每条记录被选中的机会均等。一种方法是打乱记录，选择 0~N–1 的一个随机数，其中 N 是记录数，并发射在该索引处的记录。然后再次打乱，选择 0~N–2 的随机数，依此类推。

> 统计学家可能倾向于采用**蓄水库抽样**（reservoir sampling）技术，它能够在单进程上下文中对数据流或大型数据集进行高效采样。一般来说，当你在 mapper 中使用任何概率分布时，你都必须小心，因为不能保证 mapper 在多次运行中看到相同的数据，也不能保证每个 mapper 获得相同数量的数据，更不能保证映射过程保持一个特定的顺序。这些不确定性会导致一些 mapper 预期的可能性过高或过低。对此（正确）的反应应该是将工作移动到一个 reducer 或聚合上进行，但这样做的话，在集群上多进程执行的优势可能会不复存在！虽然有分布式的蓄水库抽样算法，但是要谨记，同一算法的串行和并行实现往往可能迥然不同！

为了并行化打乱（shuffle）方法，我们可以想象将一幅扑克牌平均发给了四位玩家。如果想要抽取 4 张牌，并且让每张牌被选中的可能性相等，可以简单地让每位玩家洗他们各自手中的牌，然后每个人给你发 4 张牌；然后，你从这 16 张牌中选择前 4 张。但如果你把牌从空中掷给每位玩家，而不是平均发给他们，则每位玩家得到的牌数可能不均等，但这种方法仍然能确保每张牌被选中的可能性相等。这时问题就变成了：该如何使用 Hadoop 来打乱记录，以便获得更好的性能？

答案是在 mapper 中为每个记录分配一个 0~1 的随机浮点数。随后，mapper 会发射前 `n` 个记录。同样，reducer 也只会发射它从 mapper 接收的前 `n` 个记录。虽然此机制仍然只允许单个 reducer，但是该 reducer 获取的是数据的有限子集（例如 mapper 数量的 `n` 倍），子集应该能够放入 reducer 的内存中。因为每行拥有 `n` 个最大随机数之一的概率相等，所以能获得一个随机样本：

```
import random, heapq

class SampleMapper(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(SampleMapper, self).__init__(*args, **kwargs)

    def map(self):
        # 将堆初始化成一个包含n个0的列表
        heap = [0 for x in xrange(self.n)]

        for value in self:
            # 维护一个堆,只包含最大的n个值
            heapq.heappushpop(heap, (random.random(), value))
            for item in heap:
                # 发射抽样数据
                self.emit(None, item)


class SampleReducer(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(SampleReducer, self).__init__(*args, **kwargs)
```

```
    def reduce(self):
        # 将堆初始化为一个包含n个0的列表
        heap = [0 for x in xrange(self.n)]

        for _, values in self:
            for value in values:
                heapq.heappushpop(heap, make_tuple(value))

        for item in heap:
            # 发射抽样数据
            self.emit(None, item[1])
```

我们本可以像使用 top n 记录方法时一样使用 `bisect` 模块，但是为了获取多样性，我们使用堆数据结构在内存中维持一个只有 `n` 个最大随机值的列表。这进一步减小了 mapper 和 reducer 的内存需求（对 reducer 尤为明显），使每次只有 `n` 个值保存在内存中。我们的 mapper（reducer 也类似）初始化了一个长度 `n` 值为零的列表。`heapq.heappushpop` 函数将新值压入到堆中，然后弹出最小值（而且还比顺序调用 `heapq.push` 和 `heapq.pop` 快得多）。

3. 布隆过滤

布隆过滤器是一种高效的概率型数据结构，用于执行集合成员资格测试。布隆过滤器与其他评估函数没有什么不同，除了一点：它必须进行预先计算以收集“热值”（排除集的成员）——需要过滤的值。布隆过滤器的好处在于，它很紧凑（方便将大集合传输到集群上的每个 mapper），并且能快速测试成员资格。

但是，布隆过滤器可能会误判（false positive）；换句话说，它会把不属于集合的元素判断为属于集合。不过，它能保证不会排除任何属于集合的元素——没有漏报（false negative）。因此，表达式 `x in bloom` 就意味着“`x` 可能在集合中”或“`x` 绝对不在集合中”。这也决定了布隆过滤器的构造，因为你要在过滤器集合的大小、数据中可能有多少元素，以及 mapper 和 reducer 的内存容量之间进行权衡。通过权衡，你能采用不同程度的模糊性。在构造大多数布隆过滤器时，可以设置误判的概率阈值，这会使布隆过滤器增大或缩小。

使用布隆过滤器的第一步是构建它。布隆过滤器对输入数据应用几个散列函数，然后根据散列值设置位数组中的位。一旦构建了位数组，就可以将散列函数应用于测试数据并查看相关位是否为 1，从而测试成员资格。根据一定的规则将不同的值映射到构造布隆过滤器的 reducer，可以并行化位数组的构造过程；位数组也可以是由其他进程维护的活动的版本化数据结构。

在这个例子中，我们将使用第三方库 `pybloomfiltermmap`，可以通过 `pip` 安装。虽然 Python 有很多第三方布隆过滤器库，但这个库提供了创建可配置过滤器的最好的 API。来思考一个例子：基于推文是否包含词条和用户名白名单中的标签（#）或者 @ 回复，决定是否包含推文。为了创建布隆过滤器，从磁盘加载数据，并将布隆过滤器保存到一个 mmap 文件，如下所示：

```
from pybloomfilter import BloomFilter

bloom = BloomFilter(1000000, 0.1, 'twitter.bloom')
```

```
for prefix, path in (('#', 'hashtags.txt'), ('@', 'handles.txt')):
    with open(path, 'r') as f:
        for word in f:
            bloom.add(prefix + word.strip())
```

本示例创建了一个有 100 万元素、错误率为 0.1 的布隆过滤器。它在底层使用这些参数来选择最优数 k（所需散列函数的数量），以保证给定容量下的错误阈值。性能和空间也存在折中——容量越小且错误率越低，需要的散列函数就越多，计算也就越慢；容量越大，布隆过滤器就必须越大。从磁盘文件读取标签和 Twitter 句柄（并给它们加上适当的前缀）后，布隆过滤器将被写入磁盘上一个名为 twitter.bloom 的文件中。

在 Spark 中使用它：

```
ELEMS = re.compile(r'[#@][\w\d]+')

def tweet_filter(tweet, bloom=None):
    for elem in ELEMS.findall(tweet['text']):
        if elem in bloom.value:
            return True

# 从磁盘加载布隆过滤器,进行并行化
bloom = sc.broadcast(BloomFilter.open('twitter.bloom'))

# 从磁盘加载JSON推文,进行解析
tweets = sc.textFile('tweets').map(json.loads)
tweets = tweets.filter(partial(tweet_filter, bloom=bloom))
```

我们的推文过滤器是使用 `functools.partial` 函数创建的，该函数创建了一个拥有布隆过滤器广播变量的闭包，而布隆过滤器是从驱动程序所处主机的磁盘加载的。`tweet_filter` 函数使用一个正则表达式提取所有标签和 @ 回复，然后检查它们是否在布隆过滤器中；如果是，则返回 `True`，从而保留 RDD 中与白名单匹配的所有元素。

布隆过滤器可能是常用于 Hadoop 分析的最复杂的数据结构。此处提到它不是因为它的复杂性，而是为了表明性能和正确性的结合将如何影响分布式计算。作为实践大数据的数据科学家，你会发现随机方法能助力及时计算，这是进一步分析所需要的。

5.3　迈向最后一英里分析

在本章中，我们研究了许多数据分析模式，从键计算到聚合、过滤的常规模式。这里面有一个宏观主题：将数据从较大的输入分解为较小的、更易于管理的输入。使用我们在本章中讨论的工具，本节将讨论端到端预测模型的计算策略。

许多机器学习技术在底层使用广义线性模型（generalized linear model，GLM）来估计给定输入数据和误差分布的响应值（response variable）。最常用的 GLM 是线性回归（还有逻辑回归和泊松回归），为模拟因变量 Y 和一个或多个自变量 X 之间的连续关系建模。该关系由一组**系数**和一个**误差项**表示如下：

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n + \epsilon$$

虽然只轻描淡写地说了说这个非常重要的话题，但是必须强调的是，系数 β 的计算是将模型拟合到现有数据的主要目标。这通常通过一个优化算法来完成——根据给定的某个数据集的 X 和 Y 的观察值，该算法能找到一组最小化错误量的系数。请注意，线性回归可以被认为是一种**有监督**机器学习方法，因为“正确”答案（拟合模型的 X 和 Y 变量）是预先已知的。

普通最小二乘法和随机梯度下降这样的优化算法是迭代的；也就是说，它们要多次遍历数据。在大数据环境中，每次优化迭代都多次读取完整数据集可能会非常耗时，在按需分析或开发中尤其明显。Spark 在 MLlib 中提供的分布式机器学习算法和内存计算让情况稍有好转，具体内容将在第 9 章讨论。但如果碰上极大的数据集或极小的时间窗口，即使是 Spark 也需要花费很长时间；如果 Spark 没有你想要实现的模型或分布式算法，那么分析方法的选择范围将因分布式编程的诸多困难而受限。

通用的解决方案是贯穿整章的内容：将输入数据集转换为更小的数据集，让它可以在内存中被处理，从而达到分解问题的目的。一旦数据集被缩小成内存计算，它就可以使用标准技术进行分析，然后在整个数据集中验证。对于线性回归，我们可以对数据集进行简单随机抽样，对样本执行特征提取，构建线性模型，然后通过计算整个数据集的均方误差来验证模型。

5.3.1　模型拟合

考虑一个具体的例子：我们有一个新闻报道或博文的数据集，现在要预测接下来的 24 小时内的评论数量。针对网络爬取的原始 HTML 页面，数据流应如下所示。

(1) 解析 HTML 页面获取元数据，并将主文本与评论分开。
(2) 创建一个从时间戳到博文评论 / 评论者的索引。
(3) 使用该索引为模型创建实例，实例是一篇博文以及 24 小时滑动窗口内的评论。
(4) 将实例与主文本数据（评论和博文）连接起来。
(5) 提取每个实例的特征（例如，前 24 小时的评论数、博文长度、从窗口到发布时间之间的时间、bag of words 特征、星期几，等等）。
(6) 对实例特征取样。
(7) 使用 Scikit-Learn 或 Statsmodels 在内存中构建一个线性模型。
(8) 计算整个数据集的实例特征的均方误差或者决定系数。

该数据流表明，许多预处理作业仅需要运行一次或几次（例如，特征提取需要在整个特征分析生命周期中反复运行）。然而，模型抽样和验证过程可以例行运行。一旦启动并运行这个模型，它甚至可以在线运行，当新的信息被馈送到数据流水线中时，该模型重新进行拟合和验证。

此时，假设通过所学技术，我们已经成功得到一个具有所有特征的数据集。使用本章前面介绍过的抽样技术，可以获取更小的数据集，将其保存到磁盘，并使用 Scikit-Learn 构建一个线性模型：

```
import pickle
import numpy as np
```

```
from sklearn import linear_model

# 从磁盘的制表符分割文件加载数据
data = np.loadtxt('sample.txt')

# 目标是第一列(键),X是值
y = data[:,0]
X = data[:,1:]

# 实例化并拟合线性模型
clf = linear_model.Ridge(alpha=1.0, fit_intercept=True)
clf.fit(X, y)

# 将模型作为pickle写入磁盘
with open('clf.pickle', 'wb') as f:
    pickle.dump(clf, f)
```

这段代码使用 `np.loadtxt` 函数从磁盘加载样本数据，在这个例子中是包含实例的制表符分隔文件，第一列是目标值，其余列是特征。这种类型的输出与 Spark 和 MapReduce 将键值对写入磁盘时的格式吻合，但是你必须将集群中的数据收集到单个文件，并确保文件格式正确。然后，数据被拟合到**岭回归**，这是一种使用正则化来防止过拟合的线性回归模型。

5.3.2 模型验证

为了在集群中评估这个模型的效果，我们有两个选择。第一种，将 Scikit-Learn 线性模型属性 `clf.coef_`（系数）和 `clf.intercept_`（错误项）写入磁盘，然后将这些参数加载到我们的 MapReduce 或 Spark 作业中，并自行计算误差。然而，这需要为每个模型实现一个预测函数。第二种，使用 `pickle` 模块将模型转储到磁盘，然后将其加载到集群中的每个节点供预测使用。现在就来编写 Scikit-Learn 模型误差估计模板，因为我们在进行假设驱动开发（例如调整参数、执行特征分析或模型选择），所以可以使用任意 Scikit-Learn 模型。

要想验证模型，就必须计算整个数据集的均方误差（mean square error，MSE）。误差被定义为实际值和预测值之间的差值 $y - \hat{y}$。为了确保没有负值（这将减少误差），我们将计算平方误差的均值。为此，只需要一个计算平均值的 reducer 和一个加载模型并计算均方误差的 mapper 即可：

```
import pickle

class MSEMapper(Mapper):

    def __init__(self, model, *args, **kwargs):
        super(MSEMapper, self).__init__(*args, **kwargs)

        # 从磁盘加载模型
        with open(model, 'rb') as f:
            self.clf = pickle.load(f)


    def map(self):
        for row in self:
```

```
            # 解析行内浮点数值
            row = map(float, row)
            y = row[0]
            X = row[1:]

            yhat = self.clf.predict(x)

            self.emit(_, (y-yhat) ** 2)
```

可以在 Spark 中使用一个累加器来求平方误差之和，并将模型广播到集群上，如下所示：

```
def cost(row, clf=None):
    """
    计算给定行的平方误差
    """
    return (row[0] - clf.predict(row[1:])) ** 2


def main(sc):
    """
    Spark应用程序的主要分析过程
    """

    # 从pickle文件加载模型
    with open('clf.pickle', 'rb') as f:
        clf = sc.broadcast(model.load(f))


    # 创建累加器,求平方误差的和
    sum_square_error = sc.accumulator(0)

    # 加载和解析博客数据
    blogs = sc.textFile("blogData").map(float)

    # 映射cost函数,累加平方误差
    error = blogs.map(partial(cost, clf=clf))
    error.foreach(lambda cost: sum_square_error.add(cost))

    # 计算平均平方误差并打印
    print sum_square_error.value / error.count()
```

使用 `pickle` 模块来序列化 Scikit-Learn 模型是使用极大数据集进行机器学习的好开始。工作流通常是将经过序列化的模型存储在数据库 blob 字段中，然后根据需要在集群中进行加载和验证。更高级的大数据和扩展需要像 Mahout 和 Spark 的 MLlib 这样的机器学习库，这些内容将在第 9 章进行详细讨论。当然，对于近期开发的没有对应的分布式实现版本的模型，或者不能并行化的模型来说，我们还是有办法的。无论采用哪种方式，抽样、训练、验证策略都是行之有效的分析方法。

5.4　小结

本书以描述大数据上下文中的数据科学流程开始，重点介绍了构建数据产品和数据科学流水线。然后就顺理成章地从这些一般性话题转向了更具体的分布式计算、MapReduce 和

Spark。在本章开头，你应该已经轻松理解了分布式计算的工作原理、如何在 Hadoop 集群上实现作业，但不一定了解要实现什么。本章旨在让你了解各种分布式计算模式，并介绍了一些分析方法，以说明如何将其他数据处理工作流改编成大规模分析的版本。

我们确定的第一件事就是使用键进行计算，这是一种自然而然的并行化技巧，使我们能同时在多个集或域上进行操作。基于键的计算允许将操作同时应用到多个集合，而不需要进行若干独立的查询。理解如何使用键计算对于理解 MapReduce 非常重要，与 Spark 计算也联系紧密。为此，给 mapper 和 reducer 引入几个基于键的模式，包含 MapReduce 和 Spark 两种实现。然后，本章继续讨论更高级的算法和操作的设计模式；研究了概要、索引和过滤模式，但在这个过程中提出了非常常见的分析，如 TF-IDF、描述和随机抽样。这些模式和算法呈现了怎样用 Hadoop 进行分析，而不是怎样使用 Hadoop 进行计算。最后，展示了一个使用“最后一英里计算”计算线性回归的端到端分析的案例。我们描述了分解输入域、执行内存计算、在集群中验证计算的基本初始策略，这种策略可应用于数据流水线的许多部分，在敏捷、假设驱动的开发工作流中支持各种分析。

本章是本书第一部分和第二部分之间承上启下的部分。上半部分讨论了 Hadoop 裸机（bare metal）和计算细节，下半部分更多地将 Hadoop 看作是一个数据管理工具。接下来的章节将重点介绍 Hadoop 生态系统和支持集群中数据流水线的工具。我们将研究使用 Hive 的数据挖掘和数据仓储、使用 Sqoop 的数据采集、使用 Pig 和 Spark 等高阶工具的数据流，以及使用 Spark MLlib 的机器学习。这一章是桥梁，连通从使用 Hadoop 裸机能实现的功能，到使用这些库可能实现的功能之间的路。

第二部分

大数据科学的工作流和工具

本书第二部分将探索更高级的工作流和工具，供数据科学家使用。虽然掌握了 Hadoop、MapReduce 和 Spark 的基础知识才能了解可以进行什么样的大规模分析，但是与大数据打交道的数据科学家通常每天都围绕着构建于 Hadoop 之上的工具生态系统工作。总体来说，第二部分围绕第 1 章提出的数据产品流水线组织章节内容。

第 6 章讨论数据挖掘和数据仓储，并就关系型和列式数据存储及查询分别介绍 Hive 和 HBase；第 7 章阐明对能将数据采集到 HDFS 的采集工具的需求，研究如何使用 Sqoop 采集结构化数据，以及如何使用 Flume 采集非结构化数据；第 8 章探讨用于分析的高级 API：Apache Pig 和 Spark DataFrame；第 9 章讨论使用 Spark MLlib 的机器学习和计算方法；最后，第 10 章对以上各章讨论过的工作流进行总结，并对数据科学进行全面回顾。

第 6 章

数据挖掘和数据仓储

作为数据分析师，我们通常更愿意专注于能获取有意义见解的数据挖掘任务，或者对经过整理（curated）、清洗和分段（staging）的数据应用预测建模方法。然而，在大多数传统企业的数据环境中，在进行任何有意义的数据分析之前，都需要投入大量的工程和技术资源来收集这些数据，并将其组织到统一的数据仓库中。

因此，**企业数据仓库**（enterprise data warehouse，EDW）已经成为大多数企业处理和分析大规模数据的关键。然而，由于绝大多数 EDW 使用某种形式的关系数据库管理系统（relational database management system，RDBMS）作为主要的存储工具和查询引擎，因此在开展新的数据分析项目时，将在前期模式设计和 ETL 操作上花费大量精力。据估计，ETL 将占数据仓储成本、风险和实施时间的 70%~80%。[1] 这种开销导致即使是最一般的数据分析原型设计或探索性分析也成本高昂。

当我们需要存储和分析的数据的数据类型急剧增加时——这些数据可以是非结构化的（电子邮件、多媒体文件）或半结构化的（点击流式数据）——RDBMS 就暴露了另一个局限性：数据的速度和多样性常常需要“即时”地演进模式，这要被传统的 DW 支持是件非常困难的事。

正是出于这些原因，Hadoop 成为了数据仓储和数据挖掘领域最具革命性的技术。它将存储与处理分离，使公司能够将其原始数据存储在 HDFS 中，而不需要通过 ETL 将数据整合到一个统一的数据模型中。此外，通过使用 YARN 的通用处理层，我们能够从多个角度直接访问和查询原始数据，还能根据特定用例使用不同的方法（SQL、非 SQL）。因此，Hadoop 不仅支持探索性分析和数据挖掘原型设计，还为数据和分析的新类型打开了大门。

注 1：Kimball Group,“New Directions for ETL”(https://channels.theinnovationenterprise.com/articles/new-directions-for-etl).

本章将介绍一些 Hadoop 中的主要框架和工具，用于实现数据仓储和数据挖掘功能；还将探索 Hadoop 最受欢迎的基于 SQL 的查询引擎 Hive，以及 NoSQL 数据库 HBase；最后，将再简单介绍一些数据仓储领域的著名 Hadoop 项目。

6.1 Hive结构化数据查询

Apache Hive 是一个建立在 Hadoop 之上的“数据仓储”框架。Hive 为数据分析人员提供了熟悉的、基于 SQL 的 Hadoop 接口，使他们能为 HDFS 中的数据添加结构化模式，并能使用 SQL 查询访问和分析该数据。Hive 使熟练使用 SQL 的开发人员能发挥 Hadoop 的可扩展性和弹性，而不需要他们学习 Java 或原生的 MapReduce API。

Hive 提供了自己的 SQL 方言，被称为 Hive 查询语言（Hive Query Language，HQL）。HQL 支持许多常用的 SQL 语句，包括数据定义语句（data definition statement，DDL，例如 `CREATE DATABASE/SCHEMA/TABLE`）、数据操作语句（data manipulation statement，DMS，例如 `INSERT`、`UPDATE` 和 `LOAD`）和数据检索查询（例如 `SELECT`）。Hive 还支持集成用户定义函数，这些函数可以由 Java 或 Hadoop Streaming 支持的任何语言编写，扩展了 HQL 的内置功能。

Hive 命令和 HQL 查询被编译成**执行计划**或一系列 HDFS 操作和 / 或 MapReduce 作业，然后在 Hadoop 集群上执行。因此，Hive 继承了 HDFS 和 MapReduce 的某些限制，无法提供传统数据库管理系统应有的关键联机事务处理（online transaction processing，OLTP）功能。具体来说，因为 HDFS 是写一次，读多次（WORM）文件系统，并且不提供就地文件更新，因此 Hive 执行起行级插入、更新或删除不是非常高效。事实上，这些行级更新最近才被 Hive 的 0.14.0 版本（https://issues.apache.org/jira/browse/HIVE-5317）支持。

此外，为了满足在集群上生成和启动编译的 MapReduce 作业所需的开销，Hive 查询需要更长的延迟；在传统 RDBMS 上几秒就能完成的小型查询在 Hive 中可能需要几分钟才能完成。

好在 Hive 提供了所有基于 Hadoop 的应用程序都应有的高可扩展性和高吞吐量。因此，它非常适用于联机分析处理（online analytical processing，OLAP）的批处理任务，处理 TB 级甚至 PB 级的超大数据集。

本节将探讨 Hive 的一些主要功能，并编写 HQL 查询以执行数据分析。我们假设你已经在伪分布式模式的 Hadoop 上安装了 Hive，Hive 的安装步骤可参见附录 B。

6.1.1 Hive命令行接口（CLI）

Hive 的安装包里有一个方便的命令行接口（command-line interface，CLI），我们将使用它与 Hive 交互，并运行 HQL 语句。从 `$HIVE_HOME` 启动 Hive CLI：

```
~$ cd $HIVE_HOME
/srv/hive$ bin/hive
```

这将启动 CLI 并引导启动 logger（如果配置了）和 Hive 历史记录文件，并最终显示 Hive CLI 提示：

```
hive>
```

使用以下命令可以随时退出 Hive CLI：

```
hive> exit;
```

通过传递文件名选项 -f 和要执行的脚本的路径，Hive 也可以直接从命令行以非交互模式运行：

```
~$ hive -f ~/hadoop-fundamentals/hive/init.hql
~$ hive -f ~/hadoop-fundamentals/hive/top_50_players_by_homeruns.hql >>
~/homeruns.tsv
```

此外，带引号的查询字符串选项 -e 让你能从命令行运行内联命令：

```
~$ hive -e 'SHOW DATABASES;'
```

可以使用 -H 标志查看 CLI 的 Hive 选项的完整列表：

```
~$ hive -H

usage: hive
 -d,--define <key=value>          Variable substitution to apply to hive
                                      commands. e.g. -d A=B or --define A=B
    --database <databasename>     Specify the database to use
-e <quoted-query-string>          SQL from command line
-f <filename>                     SQL from files
-H,--help                         Print help information
-h <hostname>                     connecting to Hive Server on remote host
   --hiveconf <property=value>    Use value for given property
   --hivevar <key=value>          Variable substitution to apply to hive
                                       commands. e.g. --hivevar A=B
-i <filename>                     Initialization SQL file
-p <port>                         connecting to Hive Server on port number
-S,--silent                       Silent mode in interactive shell
-v,--verbose                      Verbose mode (echo executed SQL to the
                                  console)
```

非交互模式为运行已存脚本提供了方便，但是 CLI 使我们能够在 Hive 中轻松地调试和迭代查询。

6.1.2　Hive查询语言

在本节中，我们将通过编写 HQL 语句，创建 Hive 数据库、将 HDFS 中的数据加载到数据库，以及查询数据进行分析。本节引用的数据可以在 GitHub 仓库的 /data 目录中找到。

1.创建数据库

在 Hive 中创建数据库与在基于 SQL 的 RDBMS 中创建数据库非常相似，使用 CREATE DATABASE 或 CREATE SCHEMA 语句：

```
hive> CREATE DATABASE log_data;
```

当 Hive 创建新数据库时，模式定义数据存储在 Hive Metastore 中。如果 Metastore 中已经有该数据库，Hive 将抛出错误。我们可以通过使用 IF NOT EXISTS 来检查数据库是否存在：

```
hive> CREATE DATABASE IF NOT EXISTS log_data;
```

然后运行 SHOW DATABASES 来验证数据库是否已被创建。Hive 将返回在 Metastore 中找到的所有数据库，以及默认的 Hive 数据库：

```
hive> SHOW DATABASES;
OK
default
log_data
Time taken: 0.085 seconds, Fetched: 2 row(s)
```

此外，可以使用 USE 命令设置工作数据库：

```
hive> USE log_data;
```

这样就可以在 Hive 中创建一个数据库。可以通过在数据库中创建表定义，描述数据的结构。

2.创建表

Hive 提供了一个类似 SQL 的 CREATE TABLE 语句，最简单的形式由一个表名和一个列定义构成：

```
CREATE TABLE apache_log (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING
);
```

但是由于 Hive 数据存储在文件系统，通常在 HDFS 或本地文件系统中，所以 CREATE TABLE 命令还使用可选子句指定行格式，使用 ROW FORMAT 子句告诉 Hive 如何读取文件中的每一行并映射到我们的列。例如，可以指明数据位于由制表符分隔字段的文件中：

```
hive> CREATE TABLE shakespeare (
    lineno STRING,
    linetext STRING
)
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t';
```

Apache 访问日志的每行根据通用日志格式（https://httpd.apache.org/docs/1.3/logs.html#common）进行结构化。好在 Hive 为我们提供了一种方法，能将正则表达式应用于已知格式的记录，从而将每行**反序列化**或解析为各个组成字段。我们将使用 Hive 的 serializer-deserializer 行格式选项 SERDE 和 RegexSerDe 库来指定反序列化，并将字段映射到表列的正则表达式。需要手动将 lib 文件夹中的 hive-serde JAR 添加到当前 hive 会话，以便使用 RegexSerDe 包：

```
hive> ADD JAR /srv/hive/lib/hive-serde-0.13.1.jar;
```

现在删除之前创建的 apache_log 表，并使用自定义序列化器重新创建它：

```
hive> DROP TABLE apache_log;

hive> CREATE TABLE apache_log (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES ("input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]
*\\])([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\".*\") ([^ \"]
*|\".*\"))?", "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s
%8$s %9$s")
STORED AS TEXTFILE;
```

创建之后就可以使用 DESCRIBE 来验证表定义：

```
hive> DESCRIBE apache_log;
OK
host                    string                  from deserializer
identity                string                  from deserializer
user                    string                  from deserializer
time                    string                  from deserializer
request                 string                  from deserializer
status                  string                  from deserializer
size                    string                  from deserializer
referrer                string                  from deserializer
agent                   string                  from deserializer
Time taken: 0.553 seconds, Fetched: 9 row(s)
```

请注意，在这个表中，所有列都使用 Hive 原始数据类型 string 定义。Hive 支持 SQL 用户熟悉的许多其他原始数据类型，并且通常与 Java 支持的原始类型（https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL）对应。表 6-1 列举了这些原始数据类型。

表6-1：Hive的原始数据类型

类型	描述	示例
TINYINT	8 位带符号整数，从 −128 到 127	127
SMALLINT	16 位带符号整数，从 −32 768 到 32 767	32767
INT	32 位带符号整数	2147483647
BIGINT	64 位带符号整数	9223372036854775807
FLOAT	32 位单精度浮点数	1.99
DOUBLE	64 位双精度浮点数	3.14159265359
BOOLEAN	true 或 false	true
STRING	最大 2GB 的字符串	hello world
TIMESTAMP	纳秒级时间戳	1400561325

除原始数据类型之外，Hive 还支持可以存储值集合的复杂数据类型，表 6-2 列举了这些类型。

表6-2：Hive的复杂数据类型

类型	描述	示例
ARRAY	有序集合，数组中元素的类型必须相同	`recipients ARRAY<email:STRING>`
MAP	无序键值对集合，键必须是原始数据类型，但值可以是任意类型	`files MAP<filename:STRING,size:INT>`
STRUCT	任意类型元素集合	`address STRUCT<street:STRING,city:STRING, state: STRING,zip:INT>`

这乍看之下可能很别扭，因为关系数据库通常不支持集合类型，而是将相关集合存储在单独的表中，以维持**第一范式**，最小化数据重复和数据不一致的风险。然而，在像 Hive 这样的通过顺序扫描磁盘来处理大量非结构化数据的大数据系统中，读取嵌入集合能大大提高检索性能。[2]

有关 Hive 支持的表和数据类型选项的完整介绍，请参见 Apache Hive 语言手册（https://cwiki.apache.org/confluence/display/Hive/LanguageManual）。

3. 加载数据

创建表和定义模式之后，就可以将数据加载到 Hive 了。请注意 Hive 和传统 RDBMS 在强化模式（schema enforcement）上有一个重要区别：Hive 不会对数据执行任何证明它是否符合表模式的验证，也不会在将数据加载到表中时执行任何转换。

传统的关系数据库通过拒绝写入不符合模式定义的数据，强制执行写时模式（schema on write）；而 Hive 对查询只强制执行读时模式（schema on read）。在读取数据文件时，如果文件结构与定义的模式不匹配，Hive 通常会为缺失的或类型不匹配的字段返回 `null` 值，并尝试从错误中恢复。读时模式初始加载的速度非常快，因为数据不以数据库的内部格式读取、解析和序列化到磁盘。加载操作纯粹是将数据文件移动到 Hive 表（https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML）中对应位置的复制 / 移动操作。

Hive 中的数据加载通过 `LOAD DATA` 命令批量完成，也可以使用 `INSERT` 命令插入另一个查询的结果完成。首先，将 Apache 日志数据文件（http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html）复制到 HDFS，然后将其加载到之前创建的表中：

```
~$ hadoop fs -mkdir statistics
~$ hadoop fs -mkdir statistics/log_data
~$ hadoop fs -copyFromLocal ~/hadoop-fundamentals/data/log_data/apache.log \
    statistics/log_data/
```

可以使用 `tail` 命令验证 apache.log 是否成功上传到了 HDFS：

```
~$ hadoop fs -tail statistics/log_data/apache.log
```

一旦文件上传到了 HDFS，就返回 Hive CLI 并使用 `log_data` 数据库：

注 2：《Hive 编程指南》，Capriolo 等人著。

```
~$ $HIVE_HOME/bin/hive

hive> use log_data;
OK
Time taken: 0.221 seconds
```

使用 LOAD DATA 命令，并指定日志文件的 HDFS 路径，将内容写入到 apache_log 表中：

```
hive> LOAD DATA INPATH 'statistics/log-data/apache.log'
OVERWRITE INTO TABLE apache_log;

Loading data to table log_data.apache_log
rmr: DEPRECATED: Please use 'rm -r' instead.
Deleted hdfs://localhost:9000/user/hive/warehouse/log_data.db/apache_log
Table log_data.apache_log stats: [numFiles=1, numRows=0, totalSize=52276758,
rawDataSize=0]
OK
Time taken: 0.902 seconds
```

LOAD DATA 是 Hive 的批量加载命令。INPATH 携带一个指向默认文件系统（本例中为 HDFS）中的路径的参数。我们还可以使用 LOCAL INPATH 来指定本地文件系统上的路径。Hive 将文件移动到仓库位置。如果使用 OVERWRITE 关键字，则目标表中的所有已有数据将被删除并由数据文件输入替换；否则，新数据将被添加到表中。

数据复制并加载后，Hive 输出了一些关于加载数据的统计信息；虽然报告的 num_rows 为 0，但你可以通过运行 SELECT COUNT 来验证实际行数（省略输出）：

```
hive> SELECT COUNT(1) FROM apache_log;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
726739
Time taken: 34.666 seconds, Fetched: 1 row(s)
```

可以看到，这个 Hive 查询运行时，实际上执行了一个 MapReduce 作业来执行聚合。在 MapReduce 作业执行后，你可以看到 apache_log 表目前有 726 739 行。

6.1.3 Hive数据分析

你已经定义了一个模式并将数据加载到了 Hive 中，现在就可以对 Hive 数据库运行 HQL 查询，从而对数据进行实际的数据分析了。在本节中，我们将编写和运行 HQL 查询，根据先前导入的 Apache 访问日志数据，确定远程流量访问的高峰月份。

1. 分组

在上一节中，我们将一份 Apache 访问日志文件加载到了名为 apache_log 的 Hive 表中，其中包含 Apache Common Log 格式（http://httpd.apache.org/docs/2.2/logs.html#accesslog）的 Web 日志数据：

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200
2326
```

考虑一个计算每个自然月访问数的 MapReduce 程序。尽管这是一个非常简单的分组计数问题，但是要实现这个 MapReduce 程序仍然需要耗费不少的精力——除了要编写 `mapper`、`reducer` 和配置作业的 `main` 函数之外，还要编译和创建 JAR 文件。但有了 Hive 的话，这个问题将与运行 SQL 的 `GROUP BY` 查询一样简单直观：

```
hive> SELECT
        month,
        count(1) AS count
      FROM (SELECT split(time, '/')[1] AS month FROM apache_log) l
      GROUP BY month
      ORDER BY count DESC;
OK
Mar 99717
Sep 89083
Feb 72088
Aug 66058
Apr 64984
May 63753
Jul 54920
Jun 53682
Oct 45892
Jan 43635
Nov 41235
Dec 29789
NULL    1903
Time taken: 84.77 seconds, Fetched: 13 row(s)
```

Hive 查询和 MapReduce 程序都要对输入进行分词，并提取月份令牌作为聚合字段。不仅如此，Hive 还提供了简洁自然的查询接口来执行分组；又因为数据被结构化为 Hive 表，所以我们可以轻松地在其他任何字段上执行其他即席查询：

```
hive> SELECT host, count(1) AS count FROM apache_log GROUP BY host
ORDER BY count;
```

除了计数之外，Hive 还支持其他聚合函数来计算数字列的总和、平均值、最小值、最大值以及方差、标准差和协方差等统计聚合。使用这些内置聚合函数时，可以通过将以下属性设置为 `true` 来提高聚合查询的性能：

```
hive> SET hive.map.aggr = true;
```

这种设置告诉 Hive 在 map 阶段执行“顶层”（top-level）聚合，这与执行 `GROUP BY` 后再进行聚合不同。但请注意，此设置将需要更多内存。[3] 在 Hive 文档的“Hive Operators and User-Defined Functions（UDFs）”（http://bit.ly/1r1RnGC）中可以找到内置聚合函数的完整列表。

Hive 还提供了轻松存储计算结果的捷径。你可以创建新的表来存储这些查询返回的结果，以便进行记录保存和分析：

```
hive> CREATE TABLE
        remote_hits_by_month
        AS
```

注 3：《Hive 编程指南》，Edward Capriolo、Dean Wampler、Jason Rutherglen 著。

```
SELECT
    month,
    count(1) AS count
FROM (
    SELECT split(time, '/')[1] AS month
    FROM apache_log
    WHERE host == 'remote'
    ) l
GROUP BY month
ORDER BY count DESC;
```

CREATE TABLE AS SELECT（CTAS）操作非常有用，它能从现有 Hive 表过滤和聚合，从而派生并构建新表。

2.聚合和连接

当在单个结构化的数据集中查询和聚合数据时，Hive 能提供一些便利，我们对此也有所涉及。但在多个数据集之上执行更复杂的聚合时，Hive 才能真正物尽其用。

在第 3 章中，我们开发了一个 MapReduce 程序，根据交通研究与创新技术管理局（RITA，http://1.usa.gov/1r1RJ09）收集的航班数据分析美国航空公司的准点情况。那一章将该准点数据集进行了归一化，让单个数据文件包含所有必需的数据；但事实上，从 RITA 网站下载的数据包括航空公司和飞机的代码，它们必须分别参照单独的查找数据集。2014 年 4 月的数据已被放入 GitHub 仓库的 data/flight_data 目录。

ontime_flights.tsv 中的准点航班数据的每一行都包含一个表示航空公司代码（如 19805）的整数值和一个表示飞机代码（如“AA”）的字符串值。航空公司代码可以与 airlines.tsv 文件中相应的代码进行连接，该文件每行包含代码和相应的描述：

```
19805    American Airlines Inc.: AA
```

同理，飞机代码可以与 carriers.tsv 中对应的代码进行连接，该文件包含代码、相应的航空公司名称和生效日期：

```
AA    American Airlines Inc. (1960 - )
```

要在 MapReduce 程序中实现这些连接，需要在 map 端加载查找表到内存中进行连接，或者在 reduce 端连接。这两种方法都需要耗费大量精力编写配置作业的 MapReduce 代码；但是通过 Hive，则可以简单地将这些附加的查找数据集加载到单独的表中，并在 SQL 查询中执行连接。

假设我们已经将数据文件上传到了 HDFS 或本地文件系统。首先，为航班数据创建一个新的数据库：

```
hive> CREATE DATABASE flight_data;
OK
Time taken: 0.741 seconds
```

然后，为准点数据和查找表定义模式并加载数据（出于可读性考虑，省略输出和添加换行）：

```
hive> CREATE TABLE flights (
        flight_date DATE,
        airline_code INT,
        carrier_code STRING,
        origin STRING,
        dest STRING,
        depart_time INT,
        depart_delta INT,
        depart_delay INT,
        arrive_time INT,
        arrive_delta INT,
        arrive_delay INT,
        is_cancelled BOOLEAN,
        cancellation_code STRING,
        distance INT,
        carrier_delay INT,
        weather_delay INT,
        nas_delay INT,
        security_delay INT,
        late_aircraft_delay INT
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> CREATE TABLE airlines (
        code INT,
        description STRING
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> CREATE TABLE carriers (
        code STRING,
        description STRING
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> CREATE TABLE cancellation_reasons (
        code STRING,
        description STRING
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/ontime_flights.tsv'
      OVERWRITE INTO TABLE flights;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/airlines.tsv'
```

```
OVERWRITE INTO TABLE airlines;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/carriers.tsv'
      OVERWRITE INTO TABLE carriers;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/
        cancellation_reasons.tsv'
      OVERWRITE INTO TABLE cancellation_reasons;
```

要获取航空公司及其各自的平均起飞延误时间列表，只需要基于航空公司代码对航班和航空公司执行 SQL `JOIN`，然后使用聚合函数 `AVG()` 计算按航空公司描述分组的平均 `depart_delay`：

```
hive> SELECT
        a.description,
        AVG(f.depart_delay)
      FROM airlines a
      JOIN flights f ON a.code = f.airline_code
      GROUP BY a.description;

AirTran Airways Corporation: FL 8.035840978593273
Alaska Airlines Inc.: AS    4.746143501305276
American Airlines Inc.: AA  10.085038790027395
American Eagle Airlines Inc.: MQ    11.048787878787879
Delta Air Lines Inc.: DL    8.149843785719728
ExpressJet Airlines Inc.: EV    15.762459814292642
Frontier Airlines Inc.: F9  12.319591084296967
Hawaiian Airlines Inc.: HA  2.872051586628203
JetBlue Airways: B6 12.090553084509766
SkyWest Airlines Inc.: OO   10.086447897294379
Southwest Airlines Co.: WN  14.722817981677437
US Airways Inc.: US 7.363223345079652
United Air Lines Inc.: UA   11.124291343587137
Virgin America: VX 9.98681228106326
Time taken: 22.786 seconds, Fetched: 14 row(s)
```

如你所见，与在 MapReduce 中执行连接相比，在 Hive 中执行连接可以显著减少编码工作量。更重要的是，我们定义的结构化 Hive 数据模式使我们能够轻松添加或更改查询。让我们来修改查询，返回按飞机分组的平均起飞延误时间：

```
hive> SELECT
        c.description,
        AVG(f.depart_delay)
      FROM carriers c
      JOIN flights f ON c.code = f.carrier_code
      GROUP BY c.description;

Aces Airlines (1992 - 2003) 9.98681228106326
AirTran Airways Corporation (1994 - )   8.035840978593273
Alaska Airlines Inc. (1960 - )  4.746143501305276
American Airlines Inc. (1960 - )    10.085038790027395
American Eagle Airlines Inc. (1998 - )  11.048787878787879
```

```
Atlantic Southeast Airlines (1993 - 2011)   15.762459814292642
Delta Air Lines Inc. (1960 - )  8.149843785719728
ExpressJet Airlines Inc. (2012 - )  15.762459814292642
Frontier Airlines Inc. (1960 - 1986)    8.035840978593273
Frontier Airlines Inc. (1994 - )    12.319591084296967
Hawaiian Airlines Inc. (1960 - )    2.872051586628203
JetBlue Airways (2000 - )   12.090553084509766
Simmons Airlines (1991 - 1998)  11.048787878787879
SkyWest Airlines Inc. (2003 - ) 10.086447897294379
Southwest Airlines Co. (1979 - )    14.722817981677437
US Airways Inc. (1997 - )   7.363223345079652
USAir (1988 - 1997) 7.363223345079652
United Air Lines Inc. (1960 - ) 11.124291343587137
Virgin America (2007 - )    9.98681228106326
Time taken: 22.76 seconds, Fetched: 19 row(s)
```

Hive 可能比较适用于这些使用场景：使用的数据集的格式是结构化的、基于表格的；要进行的计算是面向批处理的 OLAP 查询，而不是实时的、面向行的 OLTP 事务。如果想了解更多有关使用和优化 Hive 的信息，推荐你阅读以示例驱动的《Hive 编程指南》，这是一本非常优秀的图书。

6.2 HBase

在上一节中，我们了解了如何使用 Hive 对存储在 HDFS 中的大型结构化数据集执行基于 SQL 的分析。但我们也发现，虽然 Hive 在 Hadoop 中提供了一个熟悉的数据操作范式，但它并不会改变存储和处理模式，而是仍然以批处理方式使用 HDFS 和 MapReduce。

回想一下，由于 HDFS 被设计为写一次、读多次（WORM）的文件系统，因此它针对顺序读取进行了优化，处理起需要对数据进行频繁或快速的行级更新的用例效率低下。这种数据访问模式通常被称为“随机访问”，需要采用这种实时、低延迟读 / 写访问的应用程序也越来越多。以呈爆炸式增长的实时传感器和遥测应用程序为例，如 NOAA（https://www.ncdc.noaa.gov/data-access/land-based-station-data）使用的从远程观测站收集天气数据的应用程序或 NASA 用于记录来自无人驾驶飞船的数据传输的深空网络（https://solarsystem.nasa.gov/basics/bsf18-1.php）。这些应用程序必须存储和处理多个传输设备以极快的速率传输过来的大量事件数据，并在查询数据时确保数据的正确性或**一致性**。因此，对于需要对数据进行随机、实时读 / 写访问的用例，需要在标准的 MapReduce 和 Hive 之外寻找数据持久层和处理层技术。

对许多数据分析应用程序来说，使用传统的关系方法建模还存在挑战。像 Facebook 的实时分析应用程序“Insights for Websites”平台（每秒跟踪超过 20 万个事件[4]）和 StumbleUpon（http://www.stumbleupon.com）的实时推荐系统[5]，它们需要同时记录来自许多数据源的大量数据事件。这些类型的实时应用程序需要记录大量基于时间的事件，这些事件往往有许多

注 4：Alex Himel,“Building Realtime Insights” (https://www.facebook.com/note.php?note_id=10150103900258920), Facebook Engineering note, 2011.03.05.

注 5：Katie Gray,“Why We Love HBase” (http://www.stumbleupon.com/blog/why-we-love-hbase/), StumbleUpon Official Blog, 2010.11.18.

种可能的结构。数据可能以某个特定值为键（如用户），但值通常表示为任意元数据的集合。以“Like”和“Share”两个事件为例，它们需要不同的列值，如表 6-3 所示。

表6-3：非结构化事件

事件ID	事件时间戳	事件类型	用户ID	文章ID	评论	接收用户ID
1	1370139285	Like	jjones	521		
2	1370139285	Share	smith	237	This is hilarious!	342
3	1370139285	Share	emiller	963	Great article	

这些类型的数据应用程序有存储**稀疏**数据的需求。在关系模型中，行是稀疏的，但列不是；也就是说，在将新行插入表后，数据库将为每个列分配存储空间，不管该字段是否有值。然而，在数据被表示为任意字段或稀疏列的集合的应用程序中，每行可能只使用可用列的一部分，这让标准关系模式既浪费资源又别扭。

6.2.1 NoSQL与列式数据库

如今，许多现代应用程序都面临着规模和敏捷的挑战，NoSQL 数据库也因此应运而生。NoSQL 是一个广泛的概念，通常指非关系数据库，涵盖广泛的数据存储模型，包括图形数据库、文档数据库、键 / 值数据存储和列族数据库。

HBase 被归类为列族或列式数据库，模型建立在 Google 的 BigTable 架构（http://research.google.com/archive/bigtable.html）之上。这种架构让 HBase 具有如下特性：

- 随机（行级）读 / 写访问；
- 强大的一致性；
- “无模式”或灵活的数据建模。

HBase 处理数据建模的方式引入了无模式的特点，它与关系数据库处理数据建模的方式非常不同。HBase 将数据组织到包含**行**的**表**中。在表中，行由唯一的**行键**标识，行键没有数据类型，而是作为**字节数组**被存储和处理。行键与关系数据库中主键的概念类似，都被自动编入索引。在 HBase 中，表行按照行键进行排序；因为行键是字节数组，所以字符串、long 的二进制表示，乃至序列化的数据结构等几乎一切都可以作为行键。HBase 将其数据存储为键值对，所有表查找都是通过表的行键或存储记录数据的唯一标识符执行的。

一行中的数据被分组成**列族**，它们由相关列组成。你可以画出一个 HBase 表，让它包含给定人口的人口普查数据，其中每行代表一个人，通过唯一的 ID 行键进行访问；每行包含个人数据和人口信息的列族，个人数据的列族包含姓名列和地址列，人口信息的列族包含出生日期列和性别列。该示例如图 6-1 所示。

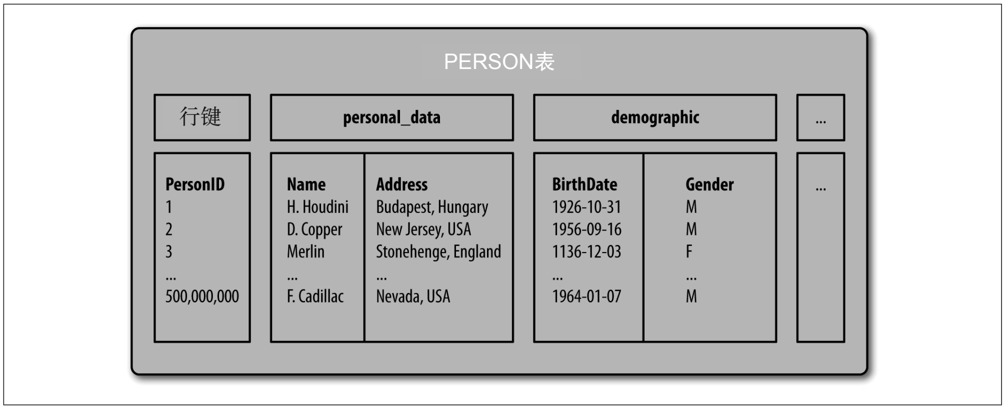


图 6-1：HBase 模式的人口普查数据

数据仓库和分析数据库的聚合都是在大量数据上进行的，这些数据可能是稀疏的，即不是所有列值都存在，因而按列存储数据比按行存储的优势更明显。尽管列族非常灵活，但列族在实践中并不完全是无模式的。在可以开始将数据插入特定行和列之前，列族就已经被定义了，因为它们会影响 HBase 存储数据的物理格局。[6] 然而，可以根据需要确定和创建组成行的实际列。事实上，每行可以包含不同的列。图 6-2 展示了一个有两行的 HBASE 表，第一个行键使用了三列族，第二个行键仅使用一列。

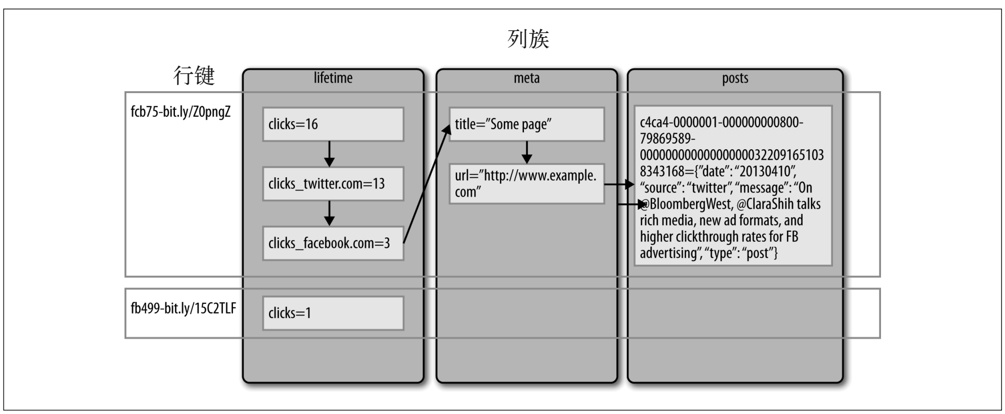


图 6-2：列稀疏的社交媒体事件

HBase 和基于 BigTable 的列式数据库还有一个有趣的特征，那就是表的单元格或行、列坐标的交集由时间戳进行版本控制；时间戳存储为一个自 1970 年 1 月 1 日 UTC 以来的长整数，用毫秒表示。因此，HBase 也被描述为一个多维的 map，其中时间提供第三维度，如图 6-3 所示。时间维度以递减顺序索引，因此从 HBase 读取时会先找到最新的值。单元格的内容可以由 `{rowkey, column, timestamp}` 元组引用，或者可以按时间范围扫描一系列单元格值。

注 6：Amandeep Khurana, “Introduction to HBase Schema Design”; login:, October 2012, Volume 37, Number 5.

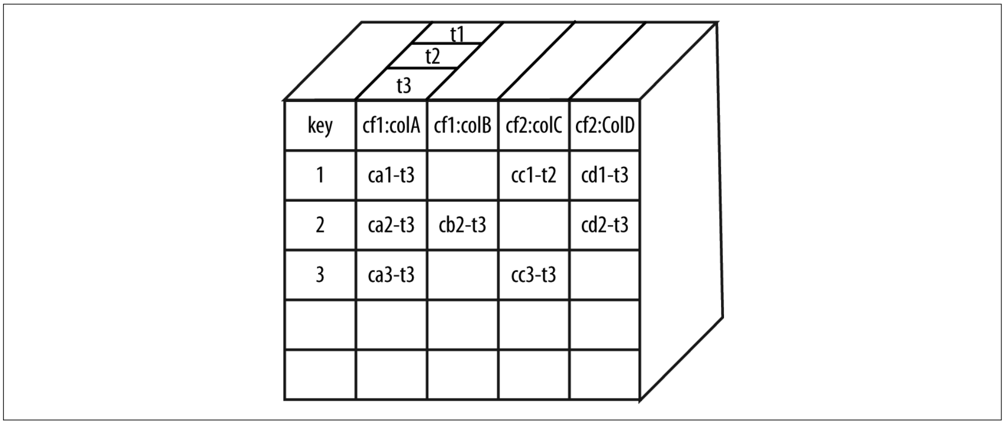


图 6-3：HBase 时间戳版本

介绍完 HBase 模式设计的主要特性，下面开始学习如何设计和查询一个简单的 HBase 表，用于假设的实时链接分享应用程序。此处假设你已经在开发环境中安装并配置了 HBase，安装和配置 HBase 的步骤见附录 B。

6.2.2　HBase实时分析

可以使用 HBase Shell 或 Java API（使用 HBaseAdmin 接口类，https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HBaseAdmin.html）创建和更新 HBase 模式。此外，HBase 还支持其他许多客户端，可用于支持非 Java 的编程语言，比如 REST API 接口、Thrift 和 Avro。[7] 这些客户端是包装了原生 Java API 的代理。

为了概述 HBase，我们定义并使用 HBase shell 设计了一个链接共享跟踪器，用来跟踪一个链接被共享的次数。但在现实情况中，你会使用原生 Java API 或受支持的客户端库编写应用程序。如果要创建外部网关客户端，请仔细考虑你的使用场景。需要高吞吐量的应用程序与纯粹的二进制格式（如 Thrift 或 Avro）更配，而 REST API 可能更适合低频率的大请求。

1.生成模式

在 HBase 中设计模式时，要着重考虑数据模型的列族结构以及它对数据访问模式的影响。定义传统关系数据库的模式主要是要准确表示实体和实体间的关系，以及考虑连接和索引等方面的性能，但成功的 HBase 模式定义往往取决于应用程序的预期用例。此外，由于 HBase 不支持连接并且只提供一个索引的行键，所以必须要小心，确保模式可以完全支持所有用例。这通常涉及使用嵌套实体的去规范化和数据重复。

幸运的是，由于 HBase 支持在运行时进行动态列定义，所以即便在创建表之后，也依然有很大的灵活性来修改和扩展模式。

注 7：参见 Apache HBase Reference Guide 中的“Apache HBase Book External APIs”（http://hbase.apache.org/book.html#external_apis）。

2.命名空间、表和列族

那么模式的哪些方面值得仔细考虑呢？首先，需要在定义表时声明表名和至少一个列族名；还可以声明自己的可选命名空间（从 Apache HBase v0.96.0 起被支持）作为表的逻辑分组，与关系数据库系统中的数据库类似。[8] 如果没有声明命名空间，HBase 将使用 `default` 命名空间：

```
hbase> create 'linkshare', 'link'
0 row(s) in 1.5740 seconds
```

我们刚刚在 `default` 命名空间中创建了一个名为 `linkshare` 的表，它有一个名为 `link` 的列族。要想在创建表后再更改表，例如更改或添加列族，需要先禁用该表，以便客户端在 `alter` 操作期间无法访问该表：

```
hbase> disable 'linkshare'
0 row(s) in 1.1340 seconds

hbase> alter 'linkshare', 'statistics'
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.1630 seconds
```

然后，使用 `enable` 命令重新启用该表：

```
hbase> enable 'linkshare'
0 row(s) in 1.1930 seconds
```

使用 `describe` 命令验证该表包含两个带有默认配置的预期列族：

```
hbase> describe 'linkshare'

Table linkshare is ENABLED
COLUMN FAMILIES DESCRIPTION
{NAME => 'link', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1',
TTL => 'FOREVER', MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
{NAME => 'statistics', DATA_BLOCK_ENCODING => 'NONE',
BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER',
KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536',
IN_MEMORY => 'false', BLOCKCACHE => 'true'}
2 row(s) in 0.1290 seconds
```

这样就创建了一个有两个列族（`link` 和 `statistics`）的 HBase 表（`linkshare`），但是这个表还不包含任何行。在插入行数据之前，需要确定如何设计行键。

3. 行键

良好的行键设计不仅会影响查询表的方式，也会影响数据访问的性能和复杂性。默认情况下，HBase 根据行键按顺序存储行，因此类似的键存储在同一个 RegionServer 中。虽然这

注 8：参见 Apache HBase Reference Guide 中的“Namespace”（http://hbase.apache.org/book.html#namespace）。

样可以更快地进行扫描范围，但在读 / 写操作期间，它也可能导致个别服务器的负载不均匀（称为“RegionServer hotspotting”）。因此，除了要实现我们的数据访问用例之外，还需要考虑各个 region 之间的行键分布。

以当前示例为例，假设我们使用唯一的反向链接 URL 作为行键。强烈推荐你阅读“Apache HBase Reference Guide” 中 的“HBase and Schema Design”（http://hbase.apache.org/0.94/book/schema.html），了解优秀的行键设计案例。

4. 使用put插入数据

现在这个表可以存储数据了——我们想在 linkshare 应用程序中存储关于链接的描述性数据（例如其标题），同时维护一个跟踪链接共享次数的频率计数器。

我们可以在指定的表 / 行 / 列和可选时间戳坐标的单元格中插入或 put 一个值。要将一个单元格值放入 linkshare 表、行键为 org.hbase.www 的行、link 列族下的当前时间戳的 title 列，可以执行以下操作：

```
hbase> put 'linkshare', 'org.hbase.www', 'link:title', 'Apache HBase'
hbase> put 'linkshare', 'org.hadoop.www', 'link:title', 'Apache Hadoop'
hbase> put 'linkshare', 'com.oreilly.www', 'link:title', 'O\'Reilly.com'
```

put 操作在插入单个单元格的值时非常有用，而对于增加频率计数器的值，HBase 提供了一种将列作为计数器来处理的特殊机制。否则，在负载较重的情况下，我们可能会遇到针对这些行的激烈竞争，因为我们需要锁定行，读取值，增加值，写回，最后再解锁该行，让其他写入过程能够访问该单元格。[9]

使用 incr 命令，而不是 put 来增加计数器的值：

```
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:share', 1
(COUNTER VALUE is now 1)

hbase> incr 'linkshare', 'org.hbase.www', 'statistics:like', 1
(COUNTER VALUE is now 1)
```

最后一个被传递的选项是增量值，在这个例子中为 1。增加计数器的值将返回更新后的计数器值，但你也可以随时使用 get_counter 命令访问计数器的当前值，指定表名、行键和列即可：

```
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:share', 1
(COUNTER VALUE is now 2)

hbase> get_counter 'linkshare', 'org.hbase.www', 'statistics:share', 'dummy'
COUNTER VALUE = 2
```

get_counter 方法用于解码计数器的字节数组值，并返回整数值。不幸的是，最新版本的 HBase 在获取计数器值的 shell 命令中引入了一个 bug——该命令的第 4 个参数没有作用。因此，需要传递哑值作为第 4 个参数：

注 9：《HBase 权威指南》，Lars George 著。

```
hbase> get_counter 'linkshare', 'org.hbase.www', 'statistics:share', 'dummy'
COUNTER VALUE = 2
```

HBase 提供了从表中检索数据的两种通用方法：get 命令通过行键执行查找，从而检索特定行的属性；scan 命令接受一组过滤器规范，并根据指定的规范进行多行迭代 。

5. 获取行或单元格值

最简单的形式是，get 命令接受表名与行键，返回行中所有列的最新版本的时间戳和单元格值：

```
hbase> get 'linkshare', 'org.hbase.www'

COLUMN                         CELL
 link:title                    timestamp=1422145743298, value=Apache HBase
 statistics:like               timestamp=1422153344211,
     value=\x00\x00\x00\x00\x00\x00\x00\x1F
 statistics:share              timestamp=1422153337498,
     value=\x00\x00\x00\x00\x00\x00\x00\x02
3 row(s) in 0.0310 seconds
```

请注意，statistics:share 列返回它的字节数组表示的值，并将每个字节打印为十六进制值。要显示计数器值的整数表示，可以使用上一节中提到的 get_counter 命令。

get 命令还接受可选的参数字典，用来指定要检索的单元格值的列、时间戳、时间范围（timerange）和版本。例如，可以指定感兴趣的列：

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => 'link:title'}
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => ['link:title',
    'statistics:share']}
```

还有一种在 get 中指定列参数的快捷方式，就是在行键之后加上以逗号分隔的列名称：

```
hbase> get 'linkshare', 'org.hbase.www', 'link:title'
hbase> get 'linkshare', 'org.hbase.www', 'link:title', 'statistics:share'
hbase> get 'linkshare', 'org.hbase.www', ['link:title', 'statistics:share']
```

为了指定感兴趣的值的时间范围，传入一个有开始时间戳和结束时间戳（以毫秒为单位）的 TIMERANGE 参数：

```
hbase> get 'linkshare', 'org.hbase.www', {TIMERANGE => [1399887705673,
    1400133976734]}
```

如果希望获取一定数量的先前版本的单元格值，而不是显式的时间戳范围，可以指定感兴趣的列，并使用 VERSIONS 参数指定要检索的版本数：

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => 'statistics:share',
    VERSIONS => 2}
```

虽然这种类型的范围查询似乎对递增为 1 的计数器值意义不大，但它为我们提供了确定分享计数器递增速率的方法，可以用它来确定链接是否为病毒。此外，这些类型的范围过滤器在执行“即时”查询时尤其有用，例如检查在一定时间范围内的指标。

6. 扫描行

scan 操作类似于数据库游标或迭代器，利用底层顺序存储机制，根据 scanner 规格迭代行数据。可以使用 scan 扫描整个 HBase 表或指定范围的行。

scan 的用法和 get 类似，它也接受 COLUMN、TIMESTAMP、TIMERANGE 和 FILTER 参数。但是，你不能指定单个行键，而是要指定一个可选的 STARTROW 和 / 或 STOPROW 参数，将扫描限制在特定的行范围内。如果不提供 STARTROW 和 STOPROW，scan 操作将扫描整个表。

事实上，你可以将表名称作为参数调用 scan，从而显示表的所有内容：

```
hbase> scan 'linkshare'

ROW                  COLUMN+CELL
com.oreilly.www      column=link:title, timestamp=1422153270279,
value=O'Reilly.com
org.hadoop.www       column=link:title, timestamp=1422153262507,
value=Apache Hadoop
org.hbase.www        column=link:title, timestamp=1422145743298,
value=Apache HBase
org.hbase.www        column=statistics:like, timestamp=1422153344211,
    value=\x00\x00\x00\x00\x00\x00\x00\x1F
org.hbase.www        column=statistics:share, timestamp=1422153337498,
    value=\x00\x00\x00\x00\x00\x00\x00\x02
3 row(s) in 0.0290 seconds
```

请记住，HBase 中的行以字典顺序存储。[10] 例如，1~100 将按如下顺序排序：

```
1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,...,9,91,92,93,94,95,96,97,98,99
```

扫描 org.hbase.www 行之后行的 link:title 列：

```
hbase> scan 'linkshare', {COLUMNS => ['link:title'], STARTROW => 'org.hbase.www'}

ROW                COLUMN+CELL
 org.hbase.www     column=link:title, timestamp=1453184861236,
 value=Apache HBase
1 row(s) in 0.0250 seconds
```

但是 STARTROW 和 ENDROW 的值不需要行键的完全匹配。它将匹配大于等于给定起始行且小于等于结束行的第一个行键；因为这些参数是包含端点的（inclusive），所以如果 ENDROW 的值与 STARTROW 相同，就不需要指定 ENDROW 了：

```
hbase> scan 'linkshare', {COLUMNS => ['link:title'], STARTROW => 'org'}

ROW                 COLUMN+CELL
org.hadoop.www      column=link:title, timestamp=1422153262507,
value=Apache Hadoop
org.hbase.www       column=link:title, timestamp=1422145743298,
value=Apache HBase
2 row(s) in 0.0210 seconds
```

注 10：参见 Apache HBase Reference Guide 中的“Data Model: Rows”（http://hbase.apache.org/book.html# datamodel）。

7. 过滤器

HBase 提供了一些过滤器类，可以进一步过滤 `get` 或 `scan` 操作返回的行数据。这些过滤器可以提供更有效的手段来限制 HBase 返回的行数据，并将行过滤操作的负担从客户端转移到服务器。HBase 可用的一些过滤器包括：

RowFilter（http://bit.ly/1r1Y7Ez）
: 基于行键值进行数据过滤。

ColumnRangeFilter（http://bit.ly/1r1Yb7m）
: 支持高效的行内扫描，可用于获取非常宽的行的**部分**列（也就是说，当一行有 100 万列时，你只想查看列 bbbb-bbdd）。

SingleColumnValueFilter（http://bit.ly/1r1Yf70）
: 基于列值过滤单元格。

RegexStringComparator（http://bit.ly/1r1Ydft）
: 用于检验给定正则表达式是否与列中的单元格值匹配。

HBase 的 Java API（https://hbase.apache.org/apidocs/）提供了一个 Filter（http://bit.ly/1r1YizN）接口、一个 FilterBase 抽象类（http://bit.ly/1r1YizN），以及一些专用的 Filter 子类（http://bit.ly/1r1YsXP）。也可以通过继承 FilterBase 抽象类并实现关键抽象方法来创建自定义过滤器。

最好在 Java 程序中使用 HBase API 来应用 HBase 过滤器，因为它们通常需要导入多个依赖的过滤器和比较器类，但是我们可以在 shell 中演示一个过滤器的简单示例。

首先，导入必需的类，包括用于将列族、列和值转换为字节的 `org.apache.hadoop.hbase.util.Bytes`、过滤器和比较器类：

```
hbase> import org.apache.hadoop.hbase.util.Bytes
hbase> import org.apache.hadoop.hbase.filter.SingleColumnValueFilter
hbase> import org.apache.hadoop.hbase.filter.BinaryComparator
hbase> import org.apache.hadoop.hbase.filter.CompareFilter
```

接下来，创建一个过滤器，筛选出 `statistics:like` 列值 ≥ 10 的行：

```
hbase> likeFilter = SingleColumnValueFilter.new(Bytes.toBytes('statistics'),
    Bytes.toBytes('like'),
    CompareFilter::CompareOp.valueOf('GREATER_OR_EQUAL'),
    BinaryComparator.new(Bytes.toBytes(10)))
```

因为不是所有行的该列都有值，所以需要设置一个标志，告诉过滤器跳过该列没有值的行：

```
hbase> likeFilter.setFilterIfMissing(true)
```

此时就可以使用配置好的过滤器运行 `scan` 操作了：

```
hbase> scan 'linkshare', { FILTER => likeFilter }

ROW                   COLUMN+CELL
org.hbase.www         column=link:title, timestamp=1422145743298,
```

```
    value=Apache HBase
org.hbase.www        column=statistics:like, timestamp=1422153344211,
    value=\x00\x00\x00\x00\x00\x00\x00\x1F
org.hbase.www        column=statistics:share, timestamp=1422153337498,
    value=\x00\x00\x00\x00\x00\x00\x00\x02
1 row(s) in 0.0470 seconds
```

这段代码应该返回 `statistics:like` 的列值≥ 10 的所有行；本示例可能包括行键 `com.oreilly.www`。

8. HBase的拓展阅读

HBase 在存储大量结构灵活的流式数据时非常有用，它还能一次查询该数据的一小块，同时确保：

- 数据“完整”（如销售或财务数据）；
- 数据会随时间变化；
- 可以查询和更新单行和部分行和列。

HBase 并不是 RDBMS、HDFS 或者 Hive 的替代品，而是提供了一种方法，在利用 Hadoop 的数据可扩展性的同时，实现对数据的随机访问。HBase 可以与传统的 SQL 或 Hive 结合，以支持需要的查询快照、范围和聚合数据。

有关使用和集成 HBase 的更多信息，建议你查阅官方的“Apache HBase Reference Guide”（http://hbase.apache.org/book.html）以及 Lars George 所著的《Hbase 权威指南》。

6.3 小结

本章介绍了 Hive 和 HBase。许多人把 Hive 当作 Hadoop 中 SQL 查询的事实标准，把 HBase 当作在 Hadoop 之上运行的、最流行的 NoSQL 数据库之一。但在深入了解 Hadoop 数据分析之后，你会发现数据仓储和数据挖掘领域中还有许多 Hadoop 项目和工具值得研究，不过这超出了本书的范围。

除了 Hive 之外，还有其他几个查询引擎也可以对 HDFS 或 HBase 进行 SQL 查询。Impala（http://impala.io）通过执行本地内存计算来提供低延迟查询，从而避免了执行 MapReduce 作业的开销；Spark SQL（https://spark.apache.org/sql/）通过运行查询（如 Spark 作业）也支持高性能 SQL 查询。

Hadoop 的优势在于，它可以灵活地支持和使用许多查询和处理引擎，选择最适合特定用例的工具。有关 Hadoop 其他数据仓储和数据挖掘解决方案（包括其他 SQL-on-Hadoop 项目）的更多信息，请参见 Mark Grover、Ted Malaska、Jonathan Seidman 和 Gwen Shapira 合著的《Hadoop 应用架构》。

第7章

数据采集

Hadoop 最突出的一个优点在于它天生就是无模式的。只要实现 Hadoop 的 Writable 接口或 DBWritable 接口并编写正确解析数据的 MapReduce 代码，Hadoop 就可以使用任何来源、任何类型或任何格式的数据，不管其结构如何（或是否缺少结构）。然而，如果输入数据驻留在关系数据库中且已结构化，使用数据已知的模式可以更方便地将数据导入 Hadoop；这种方法也比将 CSV 上传到 HDFS 并手动解析更高效。

Sqoop 用于在关系数据库管理系统（relational database management system，RDBMS）和 Hadoop 之间传输数据。它依靠 RDBMS 为要导入的数据提供模式描述，让大部分数据转换过程自动化。我们将在本章中看到，对于将关系数据库作为主要或中间数据存储的数据基础设施来说，Sqoop 是分析流程中一个非常有用的环节。

虽然用 Sqoop 将已经驻留在关系数据库中的数据大批量加载到 Hadoop 中并无不妥，但许多新的应用程序和系统涉及快速流动的数据，如应用程序日志、GPS 追踪、社交媒体动态和传感器数据。我们需要将这些数据直接加载到 HDFS 中，从而在 Hadoop 中进行处理。为了应对高吞吐量并处理由这些系统生成的基于事件的数据，就需要支持从多个源持续采集数据到 Hadoop 中。

Apache Flume 旨在从大量不同来源高效地采集、聚合和移动大量日志数据到集中的数据存储中。虽然 Flume 通常用于将流式日志数据引入 Hadoop（通常是 HDFS 或 HBase），但实际上 Flume 的数据源非常灵活，可以通过自定义向它兼容的任何消费者传输各种类型的事件数据，包括网络流量数据、社交媒体生成的数据和传感器数据。本章将介绍如何使用 Flume 从自定义日志中采集流式数据并将其传输到 Hadoop。

7.1 使用Sqoop导入关系数据

Sqoop（SQL-to-Hadoop）是由 Cloudera[1] 创建的关系数据库导入 / 导出工具，现在是 Apache 顶级项目。Sqoop 的设计初衷是在关系数据库（例如 MySQL 或 Oracle）和 Hadoop 数据存储（例如 HDFS、Hive 和 HBase）之间传输数据。它通过直接从 RDBMS 读取模式信息，自动执行大部分数据传输过程，然后使用 MapReduce 将数据导入和导出 Hadoop。[2]

Sqoop 在将数据维持在生产状态的同时，将其复制到 Hadoop 中，从而进行进一步分析，避免修改生产数据库。我们将介绍几种使用 Sqoop 将数据从 MySQL 数据库导入 Hadoop 数据存储（例如 HDFS、Hive 和 HBase）的方法。

> 本章的 Sqoop 示例假设 MySQL 数据库安装在 Sqoop 所在的主机上且可通过 localhost 访问。要安装和配置本地 MySQL 数据库，请遵循 MySQL 网站上的官方安装指南（https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/）或 Linode 网站上的简明指南（https://www.linode.com/docs/databases/mysql/install-mysql-on-ubuntu-14-04/）。请记住，大多数命令需要使用 `sudo`；而且，不要为 servername 设置主机名，因为这会在尝试通过 localhost 连接时发生冲突。

本章假设你已经安装了与你的 Hadoop 版本兼容的 Sqoop 最新稳定版本，将 Hadoop 配置为了伪分布式模式，并且所有 HDFS 和 YARN 进程也在运行中。本章将使用 MySQL 作为示例的源和目标 RDBMS，因此还假设 MySQL 数据库与 Hadoop/Sqoop 服务位于相同的主机上，可通过 localhost 和默认端口 3306 访问。安装 Sqoop 并配置 MySQL 的步骤可以在附录 B 中找到。

7.1.1 从MySQL导入HDFS

从关系数据库（如 MySQL）导入数据时，Sqoop 会从源数据库读取导入数据所需的元数据，然后提交一个仅有 map 的 Hadoop 作业，根据上一步捕获的元数据，传输实际的表数据。该作业会生成一组序列化文件，比如以符号分隔的文本文件、二进制格式（例如 Avro）文件，或包含导入的表或数据集副本的 SequenceFile 文件。[3] 默认情况下，文件以逗号分隔，并保存在 HDFS 目录中，目录名称与源表名称相对应。我们将使用这些默认设置将数据从 MySQL 导入 HDFS。

假设 MySQL 已经安装好了，继续下一步，创建一个有一些表和数据的示例数据库。首先创建一个名为 `energydata` 的数据库和一个名为 `average_price_by_state` 的表：

```
~$ mysql -uroot -p

mysql> CREATE DATABASE energydata;
```

注 1：Aaron Kimball, “Cloudera—Introducing Sqoop” (https://blog.cloudera.com/blog/2009/06/introducing-sqoop/), Cloudera Engineering Blog, June 1, 2009.

注 2：参见 Sqoop User Guide（http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html）。

注 3：参见 Sqoop User Guide 中的“Basic Usage”（http://bit.ly/1r23vHN）。

```
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON energydata.* TO '%'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON energydata.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> USE energydata;

mysql> CREATE TABLE average_price_by_state(
    year INT NOT NULL,
    state VARCHAR(5) NOT NULL,
    sector VARCHAR(255),
    residential DECIMAL(10,2),
    commercial DECIMAL(10,2),
    industrial DECIMAL(10,2),
    transportation DECIMAL(10,2),
    other DECIMAL(10,2),
    total DECIMAL(10,2)
);
Query OK, 0 rows affected (0.02 sec)

mysql> quit;
```

加载到 average_price_by_state 表中的数据由美国能源信息署（http://www.eia.gov/electricity/data/state/）提供，这些数据是 1990 年至 2012 年期间各年各州和各供应商每千瓦时（KwH）的平均电价。你可以在本书 GitHub 仓库的 /data 目录中找到名为 avgprice_kwh_state.zip 的压缩文件，其中包含名为 avgprice_kwh_state.csv 的 CSV 文件。下载该文件并将其加载到刚刚创建的 MySQL 表中：

```
~$ mysql -h localhost -u root -p energydata --local-infile=1

mysql> LOAD DATA LOCAL INFILE
    '/home/hadoop/hadoop-fundamentals/data/avgprice_kwh_state.csv'
    INTO TABLE average_price_by_state
    FIELDS TERMINATED BY ','
    LINES TERMINATED  BY '\n' IGNORE 1 LINES;

Query OK, 3272 rows affected, 6 warnings (0.03 sec)
Records: 3272  Deleted: 0  Skipped: 0  Warnings: 6

mysql> quit;
```

在运行 Sqoop 的 import 命令之前，使用 jps 命令验证 HDFS 和 YARN 是否已启动：

```
~$ sudo su hadoop
hadoop@ubuntu:~$ jps

4051 NodeManager
31134 Jps
3523 DataNode
3709 SecondaryNameNode
3375 NameNode
3921 ResourceManager
```

此时，使用 `import` 命令将表 `average_price_by_state` 中的数据导入 HDFS。可以分别通过 `--connect` 选项、`--username` 选项和 `--table` 选项来指定源数据库的连接字符串、用户名和表名。将可选的 `-m` 标志设置为 1，表示该作业使用单个 map 任务：

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost:3306/energydata
    --username root --table average_price_by_state -m 1

15/01/20 22:47:35 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
15/01/20 22:47:35 INFO manager.MySQLManager: Preparing to use a MySQL
    streaming resultset.
15/01/20 22:47:35 INFO tool.CodeGenTool: Beginning code generation
15/01/20 22:47:36 INFO manager.SqlManager: Executing SQL statement:
    SELECT t.* FROM `average_price_by_state` AS t LIMIT 1

(output truncated)

15/01/25 22:47:53 INFO mapreduce.ImportJobBase: Transferred 200.4287 KB in
    15.3718 seconds (13.0387 KB/sec)
15/01/25 22:47:53 INFO mapreduce.ImportJobBase: Retrieved 3272 records.
```

在本例中，因为表不包含主键，而主键是分割和合并多个 map 任务所必需的，所以需要指定 `import` 命令使用单个 map 任务。因为指定了导入任务使用一个 map 任务，所以 HDFS 中只有一个文件：

```
/srv/sqoop$ hadoop fs -head average_price_by_state/part-m-00000 | head

2012,AK,Total Electric Industry,17.88,14.93,16.82,0.00,null,16.33
2012,AL,Total Electric Industry,11.40,10.63,6.22,0.00,null,9.18
2012,AR,Total Electric Industry,9.30,7.71,5.77,11.23,null,7.62
2012,AZ,Total Electric Industry,11.29,9.53,6.53,0.00,null,9.81
2012,CA,Total Electric Industry,15.34,13.41,10.49,7.17,null,13.53
2012,CO,Total Electric Industry,11.46,9.39,6.95,9.69,null,9.39
2012,CT,Total Electric Industry,17.34,14.65,12.67,9.69,null,15.54
2012,DC,Total Electric Industry,12.28,12.02,5.46,9.01,null,11.85
2012,DE,Total Electric Industry,13.58,10.13,8.36,0.00,null,11.06
2012,FL,Total Electric Industry,11.42,9.66,8.04,8.45,null,10.44
```

我们现在已经成功将数据从 MySQL 导入 HDFS 了！至此，就可以对导入的数据进行后续的 MapReduce 处理，或将数据加载到另一个 Hadoop 数据源中（例如 Hive、HBase 或 HCatalog）。

7.1.2　从MySQL导入Hive

因为关系数据库（在本例中是 MySQL）中的数据已经是结构化的，所以将这些数据导入 Hive 中与源数据库类似的模式也大有用处，尤其当你打算对数据运行关系查询时。Sqoop 提供了两种方法：一种是先将数据导出到 HDFS，再在 Hive shell 中使用 `LOAD DATA HQL` 命令将其加载到 Hive 中；另一种是使用 Sqoop 直接创建表，并将关系数据库数据加载到相应的 Hive 表中。

使用 `import` 命令，Sqoop 可以根据源数据库定义的模式生成 Hive 表，并将源数据库表中的数据加载进来。但由于 Sqoop 实际上仍然使用 MapReduce 来实现数据加载操作，因此在运行导入工具之前，必须删除所有与输出名称相同的已有数据目录：

```
/srv/sqoop$ hadoop fs -rm -r /user/hadoop/average_price_by_state
```

然后运行 Sqoop 的 `import` 命令，将数据库的 JDBC 连接字符串、表名、字段分隔符、行终止符和 `null` 字符串值传递给它：

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost:3306/energydata
    --username root --table average_price_by_state
    --hive-import --fields-terminated-by ','
    --lines-terminated-by '\n' --null-string 'null' -m 1

(output truncated)

15/01/20 00:14:37 INFO hive.HiveImport: Table default.average_price_by_state stats:
    [numFiles=2, numRows=0, totalSize=205239, rawDataSize=0]
15/01/20 00:14:37 INFO hive.HiveImport: OK
15/01/20 00:14:37 INFO hive.HiveImport: Time taken: 0.435 seconds
15/01/20 00:14:37 INFO hive.HiveImport: Hive import complete.
15/01/20 00:14:37 INFO hive.HiveImport: Export directory is empty, removing it.
```

Hive 会将 double 类型的列转换为 float 类型，并去掉所有 `NOT NULL` 字段约束。除此之外，Hive 表的结构与 MySQL 的 `average_price_by_state` 表的初始定义一模一样，名字也一样。

> 如果同一台机器上也运行着 HBase 并且设置了 `HBASE_HOME` 环境变量，那么你在运行以上命令时可能会遇到以下错误：
>
> ```
> INFO hive.HiveImport: Exception in thread "main"
>     java.lang.NoSuchMethodError:
>     org/apache/thrift/EncodingUtils.setBit(BIZ)B
> ```
>
> 这是 HBase 和 Hive 之间的 Thrift 版本冲突导致的。可以将 `HBASE_HOME` 临时设置为不存在的路径，然后在导入后重新加载 bash 配置文件，就能避免错误：
>
> ```
> /srv/sqoop$ export HBASE_HOME=/fake/path
>
> (Sqoop Hive commands)
>
> /srv/sqoop$ source ~/.profile
> ```

在本地模式下，Hive 将在它运行的文件系统中创建一个 metastore_db 目录；在上一个示例中，metastore_db 是在 SQOOP_HOME（/srv/sqoop）下创建的。打开 Hive shell 并验证表 `average_price_by_state` 是否已被创建：

```
/srv/sqoop$ hive

hive> DESC average_price_by_state;

OK
year              int
state             string
sector            string
residential       double
commercial        double
industrial        double
transportation    double
```

```
other            double
total            double
Time taken: 0.858 seconds, Fetched: 9 row(s)
```

你还可以通过运行 `COUNT` 查询来验证是否已导入了 3272 行；由于该数据集相对较小，因此也可以通过运行 `SELECT * FROM average_price_by_state` 来验证数据。将数据和模式导入 Hive 之后，就可以通过 Hive 命令行接口或其他 Hive 接口对数据进行后续分析了。

7.1.3　从MySQL导入HBase

HBase 旨在为需要实时访问行级数据的大量并发客户端处理大量数据。尽管在大多数小规模和中等规模的数据应用程序中，关系数据库也能很好地满足这一要求，但如果应用程序需要扩展性更强的存储解决方案，我们就得考虑将一些大规模和重负载的数据组件转移到分布式数据库，比如 HBase。

Sqoop 的导入工具能将数据从关系数据库导入 HBase。与 Hive 一样，有两种导入方法：一种是先导入 HDFS，然后使用 HBase CLI 或 API 将数据加载到 HBase 表中；另一种是使用 `--hbase-table` 选项指示 Sqoop 直接将数据导入 HBase 表。

在这个示例中，要转移到 HBase 的数据是一个博客统计数据表，其中每条记录都包含一个主键（由竖线分隔的 IP 地址和年份）和每个月对应的列（每一列表示该年该 IP 在该月的访问数）。你可以在本书 GitHub 仓库的 /data 目录中找到名为 weblogs.csv.zip 的压缩文件。下载并解压，然后将 CSV 文件加载到 MySQL 表中：

```
~$ mysql -u root -p

mysql> CREATE DATABASE logdata;

mysql> GRANT ALL PRIVILEGES ON logdata.* TO '%'@'localhost';

mysql> GRANT ALL PRIVILEGES ON logdata.* TO ''@'localhost';

mysql> USE logdata;

mysql> CREATE TABLE weblogs (ipyear varchar(255) NOT NULL PRIMARY KEY,
    january int(11) DEFAULT NULL,
    february int(11) DEFAULT NULL,
    march int(11) DEFAULT NULL,
    april int(11) DEFAULT NULL,
    may int(11) DEFAULT NULL,
    june int(11) DEFAULT NULL,
    july int(11) DEFAULT NULL,
    august int(11) DEFAULT NULL,
    september int(11) DEFAULT NULL,
    october int(11) DEFAULT NULL,
    november int(11) DEFAULT NULL,
    december int(11) DEFAULT NULL);

mysql> quit;

~$ mysql -u root -p logdata --local-infile=1
```

```
mysql> LOAD DATA LOCAL INFILE '/home/hadoop/hadoop-fundamentals/data/weblogs.csv'
    INTO TABLE weblogs FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n' IGNORE 1 LINES;

Query OK, 27300 rows affected (0.20 sec)
Records: 27300  Deleted: 0 Skipped: 0 Warnings: 0

mysql> quit;
```

与之前一样，需要验证 Hadoop 和 HBase 守护进程是否正在运行：

```
~$ cd $HBASE_HOME
/srv/hbase$ bin/start-hbase.sh
```

然后，运行 Sqoop 的 `import` 命令，将数据库的 JDBC 连接字符串、表名、HBase 表名、列族名称和行键名称传递给它：

```
sqoop import --connect jdbc:mysql://localhost:3306/logdata
    --table weblogs --hbase-table weblogs --column-family traffic
    --hbase-row-key ipyear --hbase-create-table -m 1

(output truncated)

15/01/20 00:33:01 INFO mapreduce.ImportJobBase: Transferred 0 bytes in
19.0716 seconds (0 bytes/sec)
15/01/20 00:33:01 INFO mapreduce.ImportJobBase: Retrieved 27300 records.
```

导入的 MapReduce 作业完成后，你应该会看到控制台消息 `INFO mapreduce.ImportJobBase: Retrieved 27300 records`。可以在 HBase shell 中使用 `list` 和 `scan` 命令验证 HBase 表和行是否已被成功导入：

```
/srv/sqoop$ cd $HBASE_HOME
/srv/hbase$ bin/hbase shell

hbase(main):001:0> list

TABLE
linkshare
weblogs
2 row(s) in 1.2900 seconds

=> ["linkshare", "weblogs"]

hbase(main):002:0> scan 'weblogs', {'LIMIT' => 50}
(output truncated)
```

我们已经使用 Sqoop 的导入工具成功将关系数据从 MySQL 导入 HDFS、Hive 和 HBase。实际上，这种工具生成了一个 Java 类，它封装了导入表的行模式。[4] 该类在导入过程中被 Sqoop 使用，但也可用于后续的 MapReduce 数据处理。因此，除了自动交换 Hadoop 和关系数据库的数据之外，Sqoop 还有助于快速开发与 Hadoop 兼容的其他数据源的处理流水线。若想获取更多有关 Sqoop 特性和功能的信息，建议阅读 Apache Sqoop User Guide（http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html）。

注 4：参见 Sqoop User Guide 中的“Basic Usage”（http://bit.ly/1r25bkq）。

7.2　使用Flume获取流式数据

Flume 的设计初衷是从多个数据流中采集和获取大量数据到 Hadoop。Flume 的一个非常常见的用例是采集日志数据，例如采集 Web 服务器上由多个应用服务器发射的日志数据，将其聚合在 HDFS 中，供后续搜索或分析使用。但是，Flume 并不仅限于简单地消费和获取日志数据源，它还可以被自定义为从任何事件源传输大量的事件数据。在这两种情况下，Flume 使我们能够**在数据写入 Hadoop 后**增量且持续地获取流式数据，而不用编写自定义的客户端应用程序将数据批量加载到 HDFS、HBase 或其他 Hadoop 数据槽中。Flume 提供了一种统一而灵活的方法，将数据从大量不同且快速流动的数据流推送到 Hadoop。

Flume 的灵活性源自其固有的可扩展式数据流架构。除灵活性以外，Flume 旨在通过其分布式架构来保持容错性和可扩展性。尽管一般推荐使用默认的"端到端"可靠性模式（保证所有接收的事件最终都能发送出去）设置[5]，但 Flume 还是提供了多种冗余和恢复机制。

我们介绍了 Flume 的总体特征，但为了了解如何构建 Flume 数据流，我们需要查看它的基本构建单元：Flume 代理。

7.2.1　Flume数据流

Flume 将从起点到目的地的数据采集路径表示为**数据流**。在数据流中，一个数据单元（又称**事件**，例如一条日志）从源流经一系列跃点（hop）到达下一个目的地（https://flume.apache.org/FlumeUserGuide.html#data-flow-model）。就连 Flume 数据流最简单的实体——Flume **代理**，也体现了数据流这一概念。Flume 代理（实际上是一个 JVM 进程）是 Flume 数据流中的一个单元，一旦外部源发出事件，事件就通过代理传播。代理由三个可配置的组件构成：**源**、**通道**和**数据槽**，如图 7-1 所示。

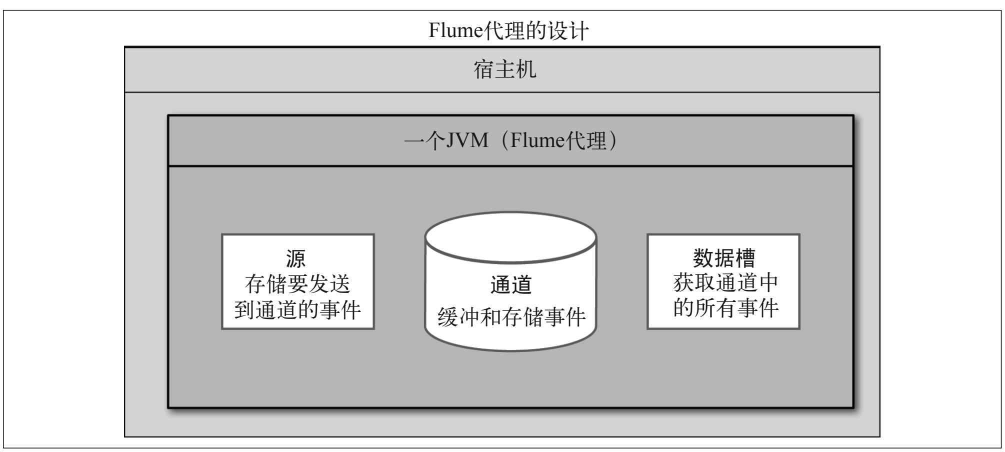


图 7-1：Flume 代理的设计

Flume 源用于监听并消费来自一个或多个外部数据源（不要与 Flume 源混淆）的事件，通

注 5：参见 Flume User Guide（https://flume.apache.org/FlumeUserGuide.html#reliability）。

过为每个数据源设置名称、类型和额外的可选参数进行配置。例如，通过运行 `tail -f /etc/httpd/logs/access_log` 命令，可以将 Flume 代理的源配置为接受来自 Apache 访问日志的事件。这种类型的源称为 `exec` 源，因为它需要 Flume 执行 Unix 命令来检索事件。

当代理消费一个事件时，Flume 源将其写入通道。该通道作为存储队列，存储和缓冲事件，直到它们可以被读取。事件以事务性方式写入通道，这意味着直到事件被消费并且相应的事务被明确地关闭，通道都将把所有事件保存在队列中。因此，即便一个代理停止，Flume 也能够保持数据事件的持久性。

Flume 数据槽最终从通道中读取和移除事件，并将其转发到下一个跃点或最终目的地。[6] 因此，可以将数据槽配置为将其输出作为流式数据源写入另一个 Flume 代理或数据存储（例如 HDFS 或 HBase）。Flume 支持多种内置数据槽，在 Apache Flume User Guide（https://flume.apache.org/FlumeUserGuide.html#flume-sinks）中均有记录。

通过使用这种源－通道－数据槽模式，可以轻松构建一个简单的 Flume 单代理数据流，从 Apache 访问日志中消费事件，并将日志事件写入 HDFS，如图 7-2 所示。

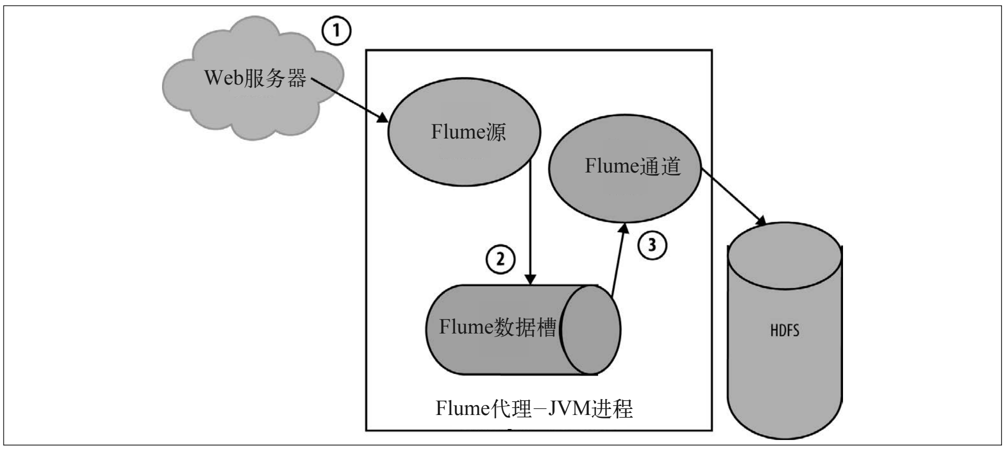


图 7-2：简单的 Flume 数据流

但是由于 Flume 代理的适配性很强，它甚至可以被配置为多个源、通道和数据槽，因此，可以通过将多个 Flume 代理链接在一起来构建多代理数据流，如图 7-3 所示。

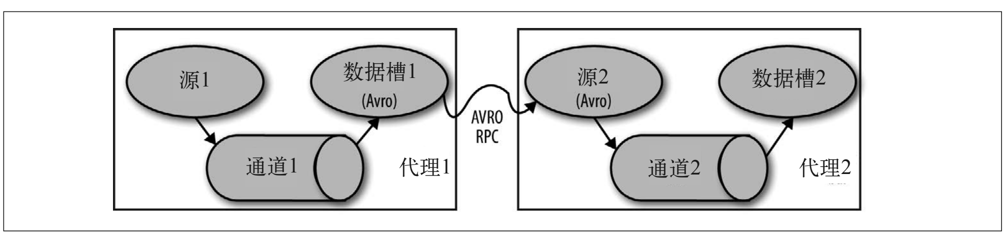


图 7-3：Flume 多代理数据流

注 6：详见《Flume：构建高可用、可扩展的海量日志采集系统》，Hari Shreedharan 著。

尽管在解决流式数据处理架构时，Flume 有一些处理常见场景的数据流拓扑模式，但是将 Flume 代理组织成复杂数据流的方式绝不仅限于此。以日志采集中的一个常见场景为例，生成日志的大量客户端将事件写入多个 Flume 代理，这被称为“第一层”代理，因为它们消费外部数据源层的数据。[7] 如果要将这些事件写入 HDFS，可以将每个第一层代理的数据槽设置为写入 HDFS；但在第一层扩展时，这可能会引起一些问题。由于几个不同的代理分别写入 HDFS，因此该数据流将无法处理周期性暴增的存储系统数据写入，从而可能引发负载和延迟的峰值。

通过添加第二层代理，可以汇总和缓冲来自第一层的事件，从而更好地将第一层代理与数据槽（HDFS）隔离。因此，第二层代理一方面可以聚合接收到的事件，降低调试难度；另一方面，它又可以控制对存储系统的写入速率，使得整个数据流可以吸收更长、更大的负载峰值。[8] 这种拓扑模式称为 fan-in 流，如图 7-4 所示。

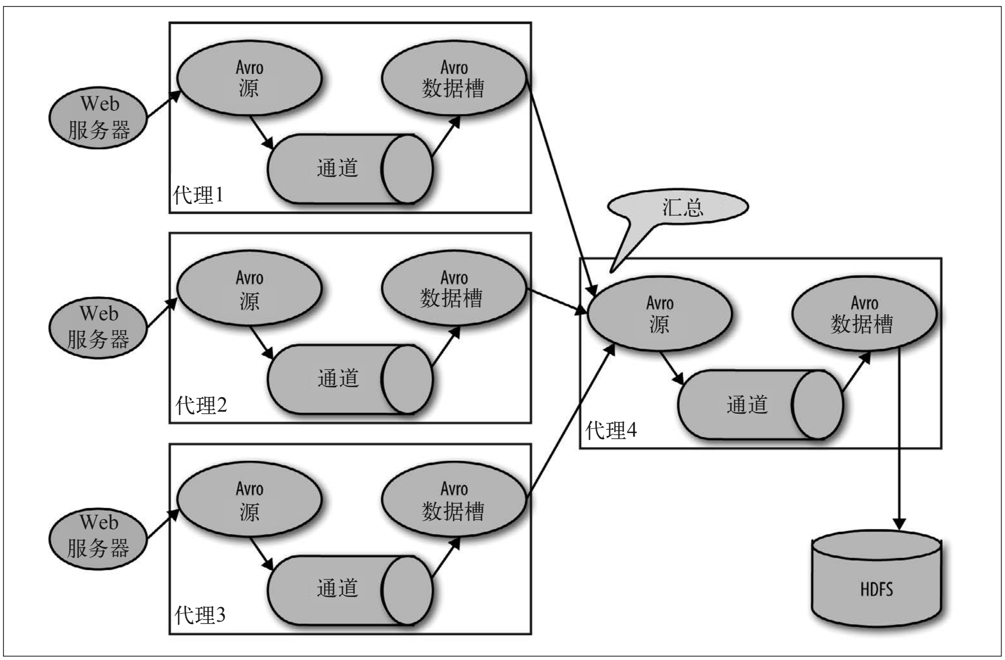


图 7-4：Flume 的 fan-in 数据流

你可能猜到了，随着产生数据的服务器的增加，第一层代理、后续代理以及层的数量通常也会相应地增加。虽然调优和设计这样复杂的 Flume 架构已经超出本书的范围，但如果你有兴趣进一步了解 Flume 的设计原则，推荐你阅读 Hari Shreedharan 的《Flume：构建高可用、可扩展的海量日志采集系统》。在下一小节中，我们将配置一个简单的 Flume 单代理数据流来获取自定义日志。

注 7：详见《Flume：构建高可用、可扩展的海量日志采集系统》，Hari Shreedharan 著。

注 8：同上。

7.2.2 使用Flume获取产品印象数据

能获取 Apache 访问日志（http://bit.ly/1r278NW）之类的标准日志数据或 Twitter firehose（http://bit.ly/1r27c04）流式数据的 Flume 单代理数据流比比皆是，但 Flume 其实也非常适用于获取自定义数据流，例如自定义应用程序生成的实时分析数据。

本小节的例子将使用 Flume 来消费某虚构在线商店生成的流式用户交互数据。许多电子商务公司都在找寻测量其在线商店微转化率的方法，追踪在线营销的效果。以下指标可用于衡量在线商店的整体效能。[9]

点击率

　产品印象转化为**点击**（或点击链接 / 图片浏览产品详情页面的操作）的次数。

加入购物车率

　点击转化为加入购物车的次数。

购物车购买率

　加入购物车转化为购买行为的次数。

购买率

　产品印象最终转化为购买行为的百分比。

通常，要获取这些指标，就需要捕获细粒度的**产品印象**，也就是在电子商务 Web 应用程序中插桩，记录访问者与产品的每个交互。这些交互包括查看产品链接、点击进入产品详情页面、向 / 从购物车添加 / 移除产品以及购买产品。在写入数据后，以一定的间隔提取和分析数据，从而生成报表、调整产品特征、驱动个性化体验等。

我们将模拟一个电子商务印象日志，以如下 JSON 格式记录用户与产品的交互信息：

```
{
    "sku": "T9921-5",
    "timestamp": 1453167527737,
    "cid": "51761",
    "action": "add_cart",
    "ip": "226.43.51.25"
}
```

动作类型包括 view、click、add_cart、remove_cart 和 purchase。在本书 GitHub 仓库的 /flume 目录中可以找到一个生成样本印象日志的脚本文件，通过执行以下操作来运行该文件：

```
$ ./impression_tracker.py
```

这将输出并创建一个名为 impressions.log 的文件，并将其放在 /tmp/impressions/ 路径下。使用拥有 sudo 权限的用户运行 setup.sh 脚本，在本地文件系统和 HDFS 中创建必需的目录：

```
$ ./setup.sh
```

注 9：Ron Kohavi and Foster Provost,“Applications of Data Mining to Electronic Commerce”, Data Mining and Knowledge Discovery 5:1–2 (2001): 5–10.

本示例模拟了一个简单的 Flume 双代理数据流，我们在其中建立了一个**客户端**代理，它在 Web 服务器上运行，获取 impressions.log 中的记录并将这些事件发送到一个 Avro 数据槽。Avro 是一种轻量级的 RPC 协议，也提供了简单的数据序列化功能。它使我们能轻松地设置 RPC 协议，将客户端代理的数据槽中的数据发送到**采集**代理的源。然后，采集代理将这些事件写入 HDFS。最终的流程如图 7-5 所示。

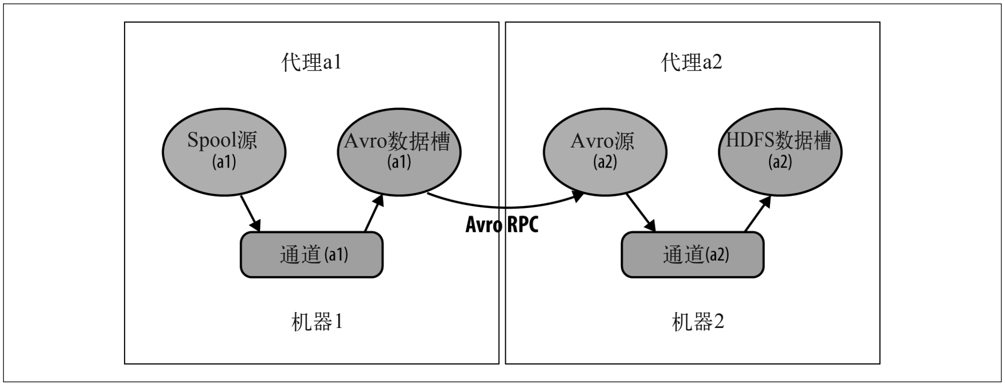


图 7-5：采集日志到 HDFS

设置 Flume 代理从编写配置文件开始。如前所述，所有 Flume 代理都由源、通道和一个或多个数据槽组成。首先将客户端代理的源配置为印象日志的位置：

```
# 定义spooling目录源
client.sources=r1
client.sources.r1.channels=ch1
client.sources.r1.type=spooldir
client.sources.r1.spoolDir=/tmp/impressions
```

将源名称设置为 `r1`，稍后将使用它来引用和设置源的其他属性。然后，为源指定一个已命名的通道，本例将其命名为 `ch1`。另外，将 `r1` 的源配置为 `spooldir` 类型，用于从磁盘指定的“假脱机”目录中获取数据。这个源将会监视指定目录中的新文件，当新文件出现时，从中解析出事件。在给定的文件被完全读入通道之后，它会被重命名，表明已被 Flume（http://bit.ly/flume-spool）完全获取。

接着，来配置客户端代理的通道，它会缓冲从源到数据槽的数据。设置一个名为 `ch1` 的通道，并将其类型设置为 `FILE`。默认情况下，文件通道通过将数据写入用户主目录下名为 checkpoint 和 data 的目录下的文件来缓存数据。可以通过配置 `checkpointDir` 和 `dataDirs` 值来覆盖特定通道的这些文件路径：

```
# 定义一个文件通道
client.channels=ch1
client.channels.ch1.type=FILE
```

最后，需要为客户端代理配置数据槽。在这个示例中，客户端代理将其数据写入 Avro 数据槽。我们将其命名为 `k1`，并将其配置为从 `ch1` 通道获取数据。Avro 数据槽需要一个主机名和端口：

```
# 定义一个Avro数据槽
client.sinks=k1
client.sinks.k1.type=avro
client.sinks.k1.hostname=localhost
client.sinks.k1.port=4141
client.sinks.k1.channel=ch1
```

接下来，配置采集代理，它从之前配置的 Avro 源消费事件，并将这些事件写入 HDFS。源、通道和数据槽的配置如下所示：

```
#定义一个Avro源
collector.sources=r1
collector.sources.r1.type=avro
collector.sources.r1.bind=0.0.0.0
collector.sources.r1.port=4141
collector.sources.r1.channels=ch1

#定义一个文件通道,为了可靠性,使用多个磁盘
collector.channels=ch1
collector.channels.ch1.type=FILE
collector.channels.ch1.checkpointDir=/tmp/flume/checkpoint
collector.channels.ch1.dataDirs=/tmp/flume/data

#定义HDFS数据槽,将事件持久化为文本
collector.sinks=k1
collector.sinks.k1.type=hdfs
collector.sinks.k1.channel=ch1
```

请注意，只要类型、绑定主机和端口的配置一致，源名称不需要与客户端代理的数据槽名称相匹配。我们也为此代理配置了一个文件通道，但是覆盖了检查点和数据目录，避免与客户端代理的通道发生冲突；此外，还声明了一个名为 `k1`、类型为 `hdfs` 的数据槽，它从为此代理配置的 `ch1` 通道消费事件。

HDFS 数据槽需要一个 `path` 配置，用于指定代理写入数据的 HDFS 位置。此外，我们还指定了其他一些可选配置参数，比如预期文件名的前缀和后缀、文件格式，以及每一批次写入的最大事件数：

```
# HDFS数据槽配置
collector.sinks.k1.hdfs.path=/user/hadoop/impressions
collector.sinks.k1.hdfs.filePrefix=impressions
collector.sinks.k1.hdfs.fileSuffix=.log
collector.sinks.k1.hdfs.fileType=DataStream
collector.sinks.k1.hdfs.writeFormat=Text
collector.sinks.k1.hdfs.batchSize=1000
```

客户端和采集代理都已完全配置好，可以运行 Flume 代理来执行完整的数据流了。首先，确保你已运行 setup.sh 脚本，并在 /tmp/impressions 下生成了 impressions.log 文件。然后，在终端打开三个选项卡。在第一个选项卡中，导航到 Flume 配置文件并运行如下命令：

```
$ flume-ng agent -n collector --conf . -f collector.conf
```

采集代理启动，等待从客户端代理接收事件。现在，在第二个选项卡中启动客户端代理：

```
$ flume-ng agent -n client --conf . -f client.conf
```

一旦客户端代理完全处理了 impressions.log 文件，你就会看到一条控制台消息，表明 impressions.log 文件已被完全处理并重命名为 impressions.log.COMPLETED：

```
INFO avro.ReliableSpoolingFileEventReader: Preparing to move file
/tmp/impressions/impressions.log to
/tmp/impressions/impressions.log.COMPLETED
```

然后检查第一个选项卡，验证采集代理是否已处理所有事件并将其写入了 HDFS。确认日志已写入配置指定的 HDFS 源目录：

```
$ hadoop fs -ls /user/hadoop/impressions/
$ hadoop fs -cat /user/hadoop/impressions/impressions.1453085307781.log | head
```

文件前缀为 impressions，后缀为 .log 扩展名，但中间的时间戳会随你运行 Flume 数据流的日期和时间而变。虽然这个双代理数据流演示的 Flume 多代理数据流非常简单，但是 Flume 为许多其他类型和配置的源、通道和数据槽提供了丰富的支持，以实现更复杂和可扩展的数据流。关于这一点，可以查阅 Flume 的用户文档（https://flume.apache.org/FlumeUserGuide.html）。

7.3　小结

在本章中，我们学习了如何使用 Sqoop 将批量数据从关系数据库高效地传输到各种 Hadoop 数据存储中。有关集成 Sqoop 的更多信息，推荐你阅读 Kathleen Ting 和 Jarek Jarcec Cecho 合著的 *Apache Sqoop Cookbook*（http://shop.oreilly.com/product/0636920029519.do）。我们还了解了 Flume 如何以可靠、可扩展的方式将流式数据传输到 Hadoop。如果你想了解更多关于 Flume 流的配置和架构的信息，推荐你阅读 Hari Shreedharan 的《Flume：构建高可用、可扩展的海量日志采集系统》。

虽然 Sqoop 和 Flume 位列 Hadoop 最常用的数据采集工具之中，但是在数据采集和流处理领域还有许多本章未涉及的 Hadoop 生态系统项目。Apache Kafka 就是其中之一，它虽然不是专门为 Hadoop 设计的，但却支持将数据高吞吐、并行地加载到 Hadoop 中。除了 Flume 和 Kafka 之外，开发采集和处理 Hadoop 和 Spark 中实时流式数据的工具和模式也是目前的研究重点。如果想进一步了解使用这些工具进行数据采集的实际用例，推荐你阅读《Hadoop 应用架构》[10] 中的“Hadoop 数据移动”一章。

注 10：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1710。——编者注

第 8 章

使用高级API进行分析

第 6 章解释了为什么要放弃原生 MapReduce，转而使用 Hive 这样的较高级语言的部分原因——因为前者实现起相对简单的操作也可能十分困难、笨拙和冗长。即便是经验丰富的 Java 和 MapReduce 程序员也会发现，大多数严谨的 Hadoop 应用程序的开发周期都很长，需要编写多个 mapper 和 reducer 并将它们链接起来，形成复杂的作业链或数据处理工作流。

此外，由于 MapReduce 旨在以批处理方式运行，因此它在运行需要响应反馈的迭代处理（包括许多机器学习算法）和交互式数据挖掘的数据分析时，会有许多限制。原生 MapReduce 在开发效率、维护和运行时表现出的性能方面的不足引发了对 Hadoop 更高层次的抽象，甚至是扩展 MapReduce 范式的新处理引擎的需求。

本章将介绍 Pig，它是 MapReduce 的一种编程抽象，有助于构建基于 MapReduce 的数据流；此外，还将介绍一些扩展核心 RDD API 的新 Spark API，让开发人员能使用他们熟悉的基于 SQL 的概念和语法，降低计算结构化数据的难度。这些项目将提供表达力强大的 API，使分析人员仅凭几行代码就能构建复杂的应用程序，从而提高 MapReduce 和 Spark 应用程序的开发效率。

8.1 Pig

和 Hive 一样，Pig 也是一种 MapReduce 抽象。它允许用户用更高级的语言去表达数据处理和分析操作，然后这些操作被编译成一个 MapReduce 作业。Pig 是雅虎开发的一个工具，通过将脚本表示为数据流[1]，使研究人员和工程师能更轻松地编写数据挖掘 Hadoop 脚本。Pig（http://pig.apache.org）现在是一个 Apache 顶级项目，包含两个主要平台组件。

注 1：《Hadoop 权威指南（第 4 版）》，Tom White 著（O'Reilly）。

- Pig Latin，用于表达数据流的过程式脚本语言。
- 运行 Pig Latin 程序的 Pig 执行环境，可以在本地或 MapReduce 模式下运行，包含 Grunt 命令行接口。

Hive 的 HQL 充分继承了 SQL 的声明式风格，但 Pig Latin 不同，它本质上是过程式的，旨在让程序员能轻松实现一系列应用于数据集的数据操作和转换，从而形成数据流水线。[2] 虽然 Hive 非常适用于能很好转换为基于 SQL 的脚本的用例 ，但要转换多重复杂数据时，SQL 就显得力不从心了。Pig Latin 是实现这类多级数据流的理想选择，特别是需要从多个源聚合数据，并在数据处理流程的每个阶段执行后续转换的情况下。

Pig Latin 脚本从数据开始，对数据应用转换，最后描述所需结果，并将整个数据处理流程作为已优化的 MapReduce 作业执行。此外，Pig 支持使用由 Java、Python、JavaScript 及其他支持的语言编写的用户定义函数（UDF）集成自定义代码。[3] 因此，我们可以使用相对简单的构造对大数据进行几乎所有的转换和临时分析。

一定要记住，Pig 和 Hive 一样，最终将编译成 MapReduce 作业，无法突破 Hadoop 批处理方法的局限性。不过 Pig 也确实为我们提供了强大的工具，可以轻松、简便地编写复杂的数据处理流程，以及在 Hadoop 上构建真实业务应用程序所需的细粒度控制。下一节将先回顾一些 Pig 的基本组件，然后通过实现原生的 Pig Latin 运算符和定制函数，对 Twitter 数据执行一些简单的情绪分析。假设你已经在伪分布式模式的 Hadoop 上安装了 Pig。安装 Pig 的步骤可以在附录 B 中找到。

8.1.1 Pig Latin

既然已经设置好了 Pig 和 Grunt shell，那就先来研究一个 Pig 脚本，探讨探讨 Pig Latin 提供的命令和表达式。以下脚本将加载一个星期内带有标签 #unitedairlines 的 Twitter 推文。

> 你可以在 GitHub 仓库的 data/sentiment_analysis/ 文件夹下找到该脚本和相应的数据。

数据文件 united_airlines_tweets.tsv 提供了 tweet ID、固定链接、发布日期、推文和 Twitter 用户名。该脚本加载字典文件 dictionary.tsv，该文件包含已知的“正面的”（positive）和“负面的”（negative）词，以及与每个词相关联的情绪评分（分别为 1 和 -1）。然后，该脚本执行一系列 Pig 变换，生成每个推文的情绪评分和分类（POSITIVE 或 NEGATIVE）：

```
grunt> tweets = LOAD 'united_airlines_tweets.tsv' USING PigStorage('\t')
    AS (id_str:chararray, tweet_url:chararray, created_at:chararray,
    text:chararray, lang:chararray, retweet_count:int, favorite_count:int,
    screen_name:chararray);
grunt> dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t')
```

注 2：Alan Gates,“Comparing Pig Latin and SQL for Constructing Data Processing Pipelines”(http://yhoo.it/1r2bK6I), Yahoo Developer Network’s Hadoop Blog, January 29, 2010.

注 3：参见 Apache Pig 的文档（http://pig.apache.org/docs/r0.12.0/start.html）。

```
    AS (word:chararray, score:int);
grunt> english_tweets = FILTER tweets BY lang == 'en';
grunt> tokenized = FOREACH english_tweets GENERATE id_str,
    FLATTEN( TOKENIZE(text) ) AS word;
grunt> clean_tokens = FOREACH tokenized GENERATE id_str,
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
grunt> token_sentiment = JOIN clean_tokens BY word, dictionary BY word;
grunt> sentiment_group = GROUP token_sentiment BY id_str;
grunt> sentiment_score = FOREACH sentiment_group
    GENERATE group AS id, SUM(token_sentiment.score) AS final;
grunt> classified = FOREACH sentiment_score
    GENERATE id, ( (final >= 0)? 'POSITIVE' : 'NEGATIVE' ) AS classification,
    final AS score;
grunt> final = ORDER classified BY score DESC;
grunt> STORE final INTO 'sentiment_analysis';
```

让我们将脚本分解成数据处理流程的单个步骤。

1. 关系和元组

脚本的前两行将数据从文件系统加载到关系 tweets 和 dictionary 中：

```
tweets = LOAD 'united_airlines_tweets.tsv' USING PigStorage('\t')
    AS (id_str:chararray, tweet_url:chararray, created_at:chararray,
    text:chararray, lang:chararray, retweet_count:int, favorite_count:int,
    screen_name:chararray);

dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t') AS (word:chararray,
score:int);
```

Pig 中的**关系**在概念上类似于关系数据库中的表，但它不是有序集合或行，而是一系列无序的元组。**元组**是一个有序的字段集合。一定要注意，虽然关系声明在赋值的左侧，与典型编程语言中的变量很像，但关系不是变量。给关系别名是为了引用，但它们其实代表的是数据处理流程中的检查点数据集。

首先，使用 LOAD 运算符指定文件的文件名（在本地文件系统或 HDFS 上），将它加载到 tweets 和 dictionary 关系中；然后，使用 USING 子句与 PigStorage 加载函数指定文件由制表符分隔；最后，使用 AS 子句定义每个关系的模式，并为每个字段指定列别名和相应的数据类型，但这步不是必需的。即使没有定义模式，仍然可以通过使用 Pig 的位置列（第一个字段为 $0，第二个字段为 $1，依此类推）引用关系中每个元组的字段。如果加载的数据有许多列，但你仅想引用其中几列时，不定义模式可能效果更好。

2. 过滤

下一行对 tweets 关系执行了简单的 FILTER 数据转换，以过滤掉所有不是英文的元组：

```
english_tweets = FILTER tweets BY lang == 'en';
```

FILTER 运算符根据某种条件从关系中选择元组，通常用它来选择所需的数据，或者过滤掉（删除）不想要的数据。因为“lang”字段类型为 chararray，而 chararray 等价于 Java 中的 String 数据类型，所以使用 == 比较运算符来筛选出值为 en（English）的记录。结果存储在一个名为 english_tweets 的新关系中。

3. 投影

过滤数据而只保留英文推文（毕竟我们的字典是英文的）之后，我们需要将推文分成单词令牌，这样就可以将单词令牌与字典进行匹配，并可对单词进行一些额外的数据清理，删除 # 和 @：

```
tokenized = FOREACH english_tweets GENERATE id_str,
FLATTEN( TOKENIZE(text) ) AS word;
clean_tokens = FOREACH tokenized GENERATE id_str,
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
```

Pig 提供了 `FOREACH ... GENERATE` 操作来处理关系或集合中的数据列，并将一组表达式应用于集合中的每个元组。`GENERATE` 子句包含值和 / 或评估表达式，评估表达式将导出一个新的元组集合并将其传递到流水线的下一步。在此示例中，我们从 `english_tweets` 关系中投影 `id_str` 键，并使用 `TOKENIZE` 函数将 `text` 字段分割成单词令牌（使用空格分割）。`FLATTEN` 函数将生成的元组集合中的每个单词提取出来，`id_str` 与其中每一个单词构成一个元组。

我们生成的元组集合实际上是 Pig 中的一种特殊数据类型，被称为 bag。它代表一个无序的元组集合，类似于关系。但关系被称为“outer bag”，因为关系不能嵌套在另一个 bag 中。在 `FOREACH` 命令中，结果产生一个叫作 `tokenized` 的新关系，它的第一个字段是 `stock_tweet` ID（`id_str`），第二个字段是由单个单词元组组成的 bag。

然后，对 `tokenized` 关系再进行投影，取得 `id_str` 和删除了开头的 # 和 @ 的小写的 `word`。我们对数据进行了很多转换，所以现在正是验证关系结构是否良好的好时机。你可以随时使用 `ILLUSTRATE` 运算符来查看基于简明样本数据集生成的每个关系的模式（为减少输出而截断了输出）：

```
grunt> ILLUSTRATE clean_tokens;
------------------------------------------------------------------
| tweets | id_str:chararray | tweet_url:chararray            |
------------------------------------------------------------------
|        | 474415416874250240 | https://.../474415416874250240 |
------------------------------------------------------------------
```

在设计 Pig 流时，定期使用 `ILLUSTRATE` 命令有助于了解查询的状态，并验证流水线中的每个检查点。

4. 分组和连接

我们已经对所选的推文进行了分词，也清洗了单词令牌，现在希望 `JOIN` 得到的令牌和字典，根据单词令牌进行匹配：

```
token_sentiment = JOIN clean_tokens BY word, dictionary BY word;
```

Pig 提供 `JOIN` 命令，基于公共字段值对两个或多个关系执行连接。虽然默认情况下使用内连接，但是内连接和外连接都是被支持的。示例基于 `word` 字段对 `clean_tokens` 关系和 `dictionary` 关系执行内连接，这将生成一个名为 `token_sentiment` 的新关系，它包含两个关系的字段：

```
-----------------------------------------------------------------------
| token_sentiment | clean_tokens::id_str:chararray |
clean_tokens::word:chararray |
dictionary::word:chararray | dictionary::score:int |
-----------------------------------------------------------------------
| | 473233757961723904 | delayedflight | delayedflight | -1 |
-----------------------------------------------------------------------
```

现在，通过 Tweet ID（即 id_str）GROUP 这些行，这样稍后才能计算每条推文的总分：

```
sentiment_group = GROUP token_sentiment BY id_str;
```

GROUP 运算符将组键（id_str）相同的元组分到一起。这个操作会产生一个关系，每个组包含一个元组，每个元组包含两个字段。

- 第一个字段名为“group”（不要将它与 GROUP 运算符弄混了），与组键的类型相同。
- 第二个字段采用原先关系的名称（token_sentiment），类型为 bag。

现在可以进行数据的最终聚合，计算按 ID 分组的每条推文的总分了：

```
sentiment_score = FOREACH sentiment_group GENERATE group AS id,
 SUM(token_sentiment.score) AS final;
```

然后根据分数，将每条推文分类为“正面”或“负面”：

```
classified = FOREACH sentiment_score GENERATE id,
( (final >= 0)? 'POSITIVE' : 'NEGATIVE' )
    AS classification, final AS score;
```

最后，将结果按分数降序排列：

```
final = ORDER classified BY score DESC;
```

至此，我们定义了情绪分析所需的所有操作和预测。下一节将把这些数据保存在 HDFS 上的一个文件中，可供之后查看、分析结果使用。

5.存储和输出数据

我们已经对数据应用了所有必要的转换，现在想在某处写出结果。为了实现这一操作，Pig 提供了 STORE 语句。它能获取某个关系，并将结果写入指定位置。默认情况下，STORE 命令将使用 PigStorage 将数据写入 HDFS 上制表符分隔的文件中。在此示例中，我们将 final 关系的结果转储到 Hadoop 用户目录（/user/hadoop/）中名为 sentiment_analysis 的文件夹中：

```
STORE final INTO 'sentiment_analysis';
```

该目录将包括一个或多个 part 文件：

```
$ hadoop fs -ls sentiment_analysis
Found 2 items
-rw-r--r-- 1 hadoop supergroup          0 2015-02-19 00:10
sentiment_analysis/_SUCCESS
-rw-r--r-- 1 hadoop supergroup       7492 2015-02-19 00:10
sentiment_analysis/part-r-00000
```

在本地模式下，只能创建一个 part 文件；但是在 MapReduce 模式下，part 文件的数量将取

决于存储前最后一个作业的并行性。Pig 提供了几个功能，能设置生成的 MapReduce 作业的 reducer 数量；你可以在 Apache Pig 文档（http://pig.apache.org/）中阅读更多有关 Pig 并行功能的内容。

当使用较小的数据集时，使用 `grunt` shell 将结果快速输出到屏幕，比将结果存储起来更方便。`DUMP` 命令使用关系的名称将其内容打印到控制台：

```
grunt> DUMP sentiment_analysis;
```

`DUMP` 命令适用于快速测试和验证 Pig 脚本的输出。但面对大型数据集的输出时，你通常会将结果存储到文件系统供后续分析使用。

8.1.2 数据类型

我们已经介绍了一些 Pig 中的可用嵌套数据结构，比如字段、元组和 bag。Pig 还提供了 map 结构，其中包含键值对集合。键始终是 `chararray` 类型，但是值的数据类型不定。在定义推文数据的模式时，我们看到了 Pig 支持的一些原生标量类型。

表 8-1 展示了所有 Pig 支持的标量类型。

表8-1：Pig标量类型

类别	类型	描述	示例
Numeric	`int`	32 位带符号整数	`12`
	`long`	64 位带符号整数	`34L`
	`float`	32 位浮点数	`2.18F`
	`double`	64 位浮点数	`3e-17`
Text	`chararray`	string 或字符数组	`hello world`
Binary	`bytearray`	blob 或字节数组	`N/A`

8.1.3 关系运算符

Pig 通过 Pig Latin 中的关系运算符提供数据操作命令。在之前的示例中，我们使用过其中几个来加载、过滤、分组、投影和存储数据。表 8-2 展示了 Pig 支持的关系运算符。

表8-2：Pig关系运算符

分类	运算符	描述
加载和存储	`LOAD`	从文件系统或其他存储源加载数据
	`STORE`	将关系存入文件系统或其他存储系统
	`DUMP`	将关系打印到控制台
过滤和投影	`FILTER`	基于某种条件从关系中选择元组
	`DISTINCT`	移除关系中重复的元组
	`FOREACH...GENERATE`	基于数据列生成数据转换
	`MAPREDUCE`	在 Pig 脚本内执行本地 MapReduce 作业
	`STREAM`	将数据发送给外部脚本或程序
	`SAMPLE`	选择一个指定大小的随机样本数据

（续）

分类	运算符	描述
分组和连接	JOIN	连接两个或多个关系
	COGROUP	将两个或多个关系的数据进行分组
	GROUP	将一个关系的数据进行分组
	CROSS	创建两个或多个关系的向量积
排序	ORDER	根据一个或多个字段对关系进行排序
	LIMIT	限制关系返回的元组数量
合并与分割	UNION	合并两个或多个关系
	SPLIT	把一个关系切分成两个或多个关系

在 Pig 的用户文档（http://pig.apache.org/docs/r0.14.0/）中可以找到 Pig 的关系运算符、算术、布尔和比较运算符的完整使用语法。

8.1.4 用户定义函数

Pig 最强大的功能之一在于，它能够让用户将 Pig 的原生关系运算符与自己的自定义处理相结合。Pig 为用户定义函数（user-defined function，UDF，http://pig.apache.org/docs/r0.14.0/udf.html）提供了广泛的支持，目前为 6 种语言提供了集成库，分别是 Java、Jython、Python、JavaScript、Ruby 和 Groovy。然而，Java 仍然是最广泛支持的编写 Pig UDF 的语言，并且通常更高效——因为它与 Pig 是相同的语言，可以与 Pig 接口（例如 `Algebraic` 接口和 `Accumulator` 接口）集成。

来演示一个之前写过的脚本的简单 UDF。在这种场景下，我们想编写一个自定义的评估 UDF，将分数分类评估转换为一个函数。这样的话，就不需要写：

```
classified = FOREACH sentiment_score GENERATE id,
    ( (final >= 0)? 'POSITIVE' : 'NEGATIVE' )
    AS classification, final AS score;
```

而是写：

```
classified = FOREACH sentiment_score GENERATE id,
    classify(final) AS classification, final AS score;
```

在 Java 中，我们需要扩展 Pig 的 `EvalFunc` 类并实现 `exec()` 方法。该方法需要一个元组，并返回一个 String：

```
package com.statistics.pig;

import java.io.IOException;

import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.Tuple;

public class Classify extends EvalFunc {

    @ Override
```

```
    public String exec(Tuple args)throws IOException {
        if (args == null || args.size() == 0) {
            return false;
        }
        try {
            Object object = args.get(0);
            if (object == null) {
                return false;
            }
            int i = (Integer)object;
            if (i >= 0) {
                return new String("POSITIVE");
            } else {
                return new String("NEGATIVE");
            }
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }
}
```

要使用此函数，就需要先编译它，将它打包成 JAR 文件，然后使用 `REGISTER` 运算符将它注册到 Pig：

```
grunt> REGISTER statistics-pig.jar;
```

然后通过命令调用该函数：

```
grunt> classified = FOREACH sentiment_score GENERATE id,
    com.statistics.pig.Classify(final) AS classification, final AS score;
```

建议你阅读 UDF 的相关文档，其中包含支持的 UDF 接口列表，并提供用于评估、加载、存储、汇总 / 过滤数据任务的示例脚本。Pig 还提供了一组由用户贡献的 UDF，叫作 Piggybank。它们与 Pig 一起发布，但必须注册才能使用。有关 Piggybank 的详细信息，请参见 Apache 文档（https://cwiki.apache.org/confluence/display/PIG/PiggyBank）。

8.1.5 Pig小结

对偏爱过程式编程模型的用户来说，Pig 是一个强大的工具。它能控制流水线中的数据检查点，更提供了对每个步骤如何处理数据的细粒度控制。当你需要更灵活地控制数据流中的操作顺序（例如提取、转换、加载或 ETL 过程），或者面对不适合使用 Hive 的 SQL 语法的半结构化数据时，Pig 都是一个很好的选择 。

8.2 Spark高级API

现在的许多项目和工具都围绕着 MapReduce 和 Hadoop 构建，以支持常见的数据任务并提供更高效的开发人员体验。例如我们已经见过的，使用 Hadoop Streaming 这样的框架以非 Java 语言（如 Python）编写和提交 MapReduce 作业。我们还介绍了为 MapReduce 提供更高级别抽象的工具——Hive 和 Pig。Hive 提供了关系型接口和声明式的基于 SQL 的语言来查询结构化数据，而 Pig 提供了一个在 Hadoop 中编写面向数据流程序的过程式接口。

但在实际工作中，典型的分析工作流将结合关系查询、过程式编程和自定义处理三者。这意味着大多数端到端 Hadoop 工作流都要集成多个不同组件，并在不同编程 API 之间切换。与以 MapReduce 为中心的 Hadoop 栈相比，Spark 在编程上有两大优势。

- 内置表达力强大的 API，以标准的通用语言（例如 Scala、Java、Python 和 R）提供。
- 包含若干内置高级库的统一编程接口，支持广泛的数据处理任务，比如复杂的交互式分析、结构化查询、流处理和机器学习。

第 4 章使用 Spark 的基于 Python 的 RDD API 编写了一个程序，在不使用工具类的大约 10 行代码中，对数据集进行了加载、清洗、连接、过滤和排序。如你所见，Spark 的 RDD API 提供了更丰富的功能操作，代码量远远小于在 MapReduce 中编写的类似程序。然而，因为 RDD 是一种通用的、与类型无关的数据抽象，而且 RDD 固定的模式只有你知道，所以处理结构化数据的过程非常繁琐；这通常将迫使你必须编写大量重复代码才能访问内部数据类型，以及将简单查询操作转换为 RDD 操作的函数语义。考虑图 8-1 所示的操作，该操作试图计算各学科教授的平均年龄。

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	42	B Jones
Chem	61	M Kennedy

RDD API

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceBy Key(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

图 8-1：使用 Spark 的 RDD API 进行聚合

在实践中，使用关系型数据的通用语言 SQL 来处理这样的结构化表格数据要更自然一些。好在 Spark 提供了一个集成的模块，让我们仅用一行简单的代码就表达了前面的聚合，如图 8-2 所示。

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	42	B Jones
Chem	61	M Kennedy

DataFrame API

```
data.groupBy(“dept”).avg(“age”)
```

图 8-2：使用 Spark 的 DataFrame API 进行聚合

8.2.1　Spark SQL

Spark SQL 是 Apache Spark 中的一个模块，它提供了一个关系型接口，让你在 Spark 中使用熟悉的基于 SQL 的操作来处理结构化数据。可以通过 JDBC/ODBC 连接器、内置的交互式 Hive 控制台或其内置的 API 访问 Spark SQL。最后一种访问方式是其中最有趣，也是最强大的，因为实际上，Spark SQL 是作为库在 Spark 的 Core 引擎和 API 之上运行的。所以，可以使用与 Spark 的 RDD API 相同的编程接口访问 Spark SQL 的 API，如图 8-3 所示。

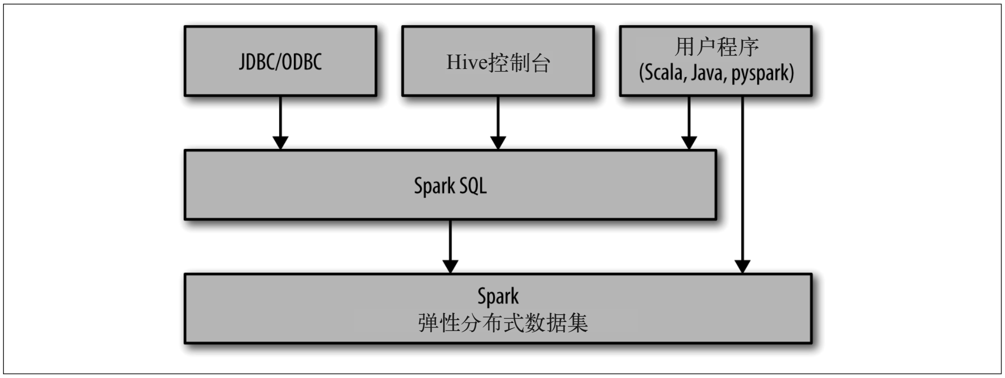


图 8-3：Spark SQL 接口

这让我们能在一个编程环境中，将关系查询的优势、Spark 过程式处理的灵活性和 Python 分析库的强大功能充分结合并付诸实践。[4]

来写一个简单的程序，用 Spark SQL API 加载 JSON 数据并进行查询。你可以在运行着的 `pyspark` shell 中直接输入这些命令，也可以在使用 `pyspark` 内核的 Jupyter notebook 中输入这些命令；无论使用哪种方法，都要确保有一个运行着的 SparkContext，因为假定变量 `sc` 将引用它。

> 以下示例使用 /sparksql 目录下运行的 Jupyter notebook。确保你已经解压了 GitHub 仓库的 /data 目录中的 sf_parking.zip 文件。可以在 GitHub 仓库的 /sparksql 目录中查看 sf_parking.ipynb 文件。

首先，从 pyspark.sql 包导入 `SQLContext` 类。`SQLContext` 类是 Spark SQL API 的入口，通过包装活动的 `SparkContext` 对象创建：

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

在此示例中，我们将从 SF Open Data（https://data.sfgov.org）加载一个 JSON 格式的数据集，该数据集列出了 2011 年 9 月旧金山公开可用的路边停车位。[5]

注 4：Michael Armbrust et al.,“Spark SQL: Relational Data Processing in Spark”, ACM SIGMOD Conference 2015.

注 5：SF Open Data,“Off-Street Parking Lots and Parking Garages”(http://bit.ly/1r2n9Do).

与 Hadoop 一样，Spark SQL 也要求将 JSON 数据格式化。所以，要删除第一个和最后一个大括号或中括号，让每个 JSON 对象都包含在一行中，后跟换行符（即没有跨越多行的 JSON 对象）。我们使用实用程序 clean_json.py 为你提供了一个已清洗的数据文件，名为 sf_parking_clean.json。

但对于极大的数据集，你可以使用 Spark 来执行格式化。例如，如果需要手动删除 JSON 文件的第一个和最后一个方括号，可以这样加载和格式化文件：

```
input = sc.wholeTextFiles(input_path).map \
(lambda (x,y): y)
data = input.flatMap(lambda x: json.loads(x))
data.map(lambda x: json.dumps(x)) \
.saveAsTextFile(output_path)
```

`wholeTextFiles` 函数创建了一个 `PairRDD`，其中键是拥有完整路径的文件名（例如“hdfs://localhost:9000/user/hadoop/sf_parking/sf_parking.json”），值是整个文件内容的字符串。使用 `map` 操作提取内容作为输入，然后使用 `flatMap` 将字符串内容读取为 JSON 格式。

调整文件至正确格式后，通过调用 `sqlContext.read.json` 并将文件的路径传递给它，就可以轻松加载文件内容了：

```
parking = sqlContext.read.json('../data/sf_parking/sf_parking_clean.json')
```

也可以将一个目录的路径传递给 `sqlContext`，`sqlContext` 会将其中所有的文件加载到 `parking` 对象中。Spark SQL 自动推断 JSON 数据集的模式，可以使用 `printSchema` 方法以漂亮的树形式来显示它。

```
parking.printSchema()

root
 |-- address: string (nullable = true)
 |-- garorlot: string (nullable = true)
 |-- landusetyp: string (nullable = true)
 |-- location_1: struct (nullable = true)
 |    |-- latitude: string (nullable = true)
 |    |-- longitude: string (nullable = true)
 |    |-- needs_recoding: boolean (nullable = true)
 |-- mccap: string (nullable = true)
 |-- owner: string (nullable = true)
 |-- primetype: string (nullable = true)
 |-- regcap: string (nullable = true)
 |-- secondtype: string (nullable = true)
 |-- valetcap: string (nullable = true)
```

还可以查看第一行数据：

```
parking.first()

Row(address=u'2110 Market St', garorlot=u'L', landusetyp=u'restaurant',
```

```
location_1=Row(latitude=u'37.767378', longitude=u'-122.429344',
needs_recoding=False), mccap=u'0', owner=u'Private', primetype=u'PPA',
regcap=u'13', secondtype=u' ', valetcap=u'0')
```

为了对数据集运行 SQL 语句，必须先将其注册为临时命名表：

```
parking.registerTempTable("parking")
```

这才允许我们运行额外的表和 SQL 方法，比如用表格形式显示前 20 行数据的 show：

```
parking.show()

...output truncated...
```

如果要在 parking 表上执行 SQL 语句的话，就需要使用 sql 方法，并将完整的查询语句传递给它。来运行一个聚合，按照主要类型和次要类型对停车场地进行分组，获得停车场地的数量以及普通停车位的平均个数。将其存储在 aggr_by_type 中，并调用 show() 来查看完整的结果：

```
aggr_by_type = sqlContext.sql("SELECT primetype, secondtype,
                               count(1) AS count,
                               round(avg(regcap), 0) AS avg_spaces " +
                              "FROM parking " +
                              "GROUP BY primetype, secondtype " +
                              "HAVING trim(primetype) != '' " +
                              "ORDER BY count DESC")

aggr_by_type.show()
```

除了 JSON 之外，Spark SQL 还支持其他几种数据源，比如本地文件系统、HDFS 或 S3 中的文件（例如文本文件、parquet 文件、CSV 文件等；CSV 文件可以使用 Databricks 的 CSV-reader 实用程序解析，https://github.com/databricks/spark-csv）、JDBC 数据源（例如 MySQL）和 Hive。此外，Spark 甚至可以作为 Hive 的底层执行引擎使用，只需在活动的 Hive 会话中设置 hive.execution.engine=spark 即可。

但 Spark SQL 模块可不仅仅是一个 SQL 接口而已，它的强大归根到底来源于其底层的数据抽象——DataFrame。

8.2.2 DataFrame

DataFrame 是 Spark SQL 中的底层数据抽象。Python Pandas（http://pandas.pydata.org）和 R（https://www.r-project.org）的用户应该非常熟悉数据框的概念；事实上，Spark 的 DataFrame 与原生的 Pandas（使用 pyspark）和 R 数据框（使用 SparkR，https://spark.apache.org/docs/1.6.0/sparkr.html）是可以互操作的。DataFrame 在 Spark 中也表示已定义模式的数据的表。Spark 的 DataFrame 和 Pandas、R 的数据框的关键区别是，前者实际上是一个包装了 RDD 的分布式集合；你可以将其视为行对象的 RDD。

此外，DataFrame 操作在底层进行了许多优化，不仅将查询计划编译为可执行代码，而且与硬编码的 RDD 操作相比，性能有显著提升，内存占用的空间也大幅减少。事实上，如果在一个基准测试中，将聚合了 1000 万整数对的 DataFrame 代码和与之等效的 RDD 代码

进行运行时间上的对比，你会发现 DataFrame 不仅速度快 4~5 倍，而且还消除了 Python 和 JVM 实现（http://bit.ly/1r2vMhm）的性能差距，如图 8-4 所示。

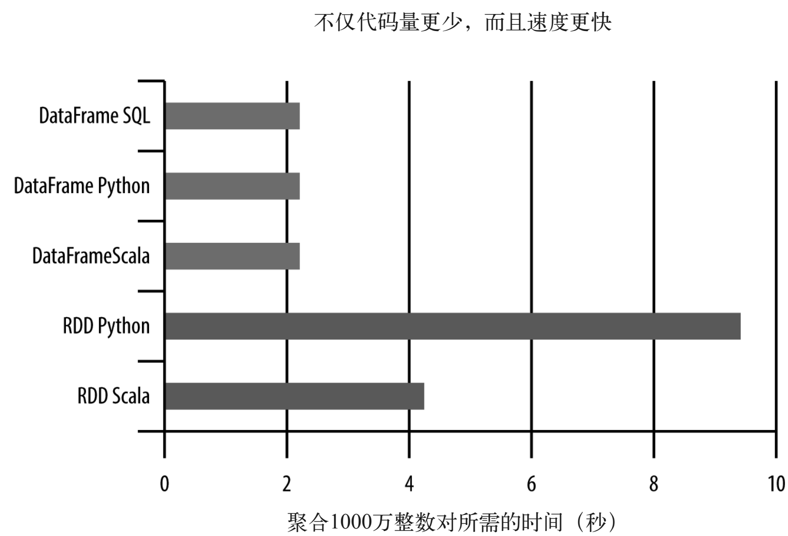


图 8-4：DataFrame 优化

DataFrame API 简洁直观的语义，加上它计算引擎提供的性能优化，促使 DataFrame 成为了 Spark 所有模块（包括 Spark SQL、RDD、MLlib 和 GraphX）的主要接口。通过这种方式，DataFrame API 提供了统一的引擎，跨越了 Spark 的所有数据源、工作负载和环境，如图 8-5 所示。

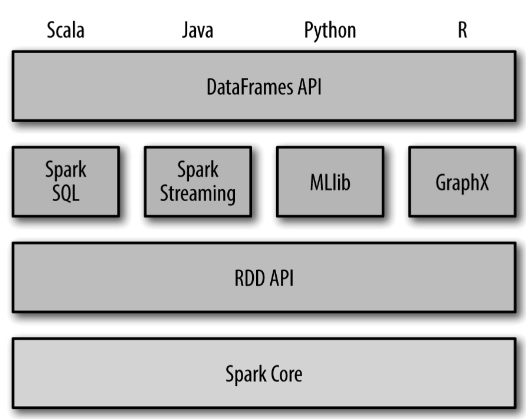


图 8-5：作为 Spark 统一接口的 DataFrame

在上一个例子中，我们使用 Spark SQL 的 `read` 接口加载了 SF 停车场地数据。但实际上，

我们创建了一个叫作 parking 的 DataFrame。在那个例子中，我们将 DataFrame 注册为临时表来执行原始 SQL 查询，在 parking DataFrame 上也有很多可以调用的关系运算符和窗口函数。事实上，通过将几个简单的 DataFrame 操作连接起来，就可以重写上一个例子中的 SQL 查询：

```
from pyspark.sql import functions as F

aggr_by_type = parking.select("primetype", "secondtype", "regcap") \
                    .where("trim(primetype) != ''") \
                    .groupBy("primetype", "secondtype") \
                    .agg(
                       F.count("*").alias("count"),
                       F.round(F.avg("regcap"), 0).alias("avg_spaces")
                    ) \
                    .sort("count", ascending=False)
```

与原始 SQL 相比，这种方法可以通过连续链接和测试操作，轻松迭代复杂查询，这是它的优势。此外，我们还可以使用 DataFrame API 访问大量的内置函数集合，比如之前使用的 count、round 和 avg 聚合函数。pyspark.sql.functions 模块还包含一些数学和统计工具，其中包含的函数可用于：

- 随机数据的生成
- 总结和描述性数据
- 样本协方差和相关性
- 交叉表（亦称列联表）
- 频率计算
- 数学函数

来使用一个这样的函数计算一些描述性的总结统计数据，以便对可用停车场地数据的分布和频率有更好的了解。describe 函数返回一个 DataFrame，其中包含每个指定数字列的非空条目计数、平均值、标准差、最小值和最大值：

```
parking.describe("regcap", "valetcap", "mccap").show()

+-------+------------------+------------------+------------------+
|summary|            regcap|          valetcap|             mccap|
+-------+------------------+------------------+------------------+
|  count|              1000|              1000|              1000|
|   mean|           137.294|             3.297|             0.184|
| stddev|361.05120902655824|22.624824279398823|1.9015151221485882|
|    min|                 0|                 0|                 0|
|    max|               998|                96|                 8|
+-------+------------------+------------------+------------------+
```

也许我们想确定停车场地所有者和停车场主要类型（“primetype”）之间的联合频率分布。在统计学中，这通常通过计算**列联表**或**交叉表**完成，这两种表以矩阵格式显示两个变量之间的共现频率。使用 stat 接口中的 crosstab 方法就可以轻松为 Spark DataFrame 计算该分布了：

```
parking.stat.crosstab("owner", "primetype").show()

+------------------+---+---+---+---+---+
|    owner_primetype|PPA|PHO|CPO|CGO|   |
+------------------+---+---+---+---+---+
|         Port of SF|  7|  7|  0|  4|  0|
|               SFPD|  0|  3|  0|  6|  0|
|              SFMTA| 42| 14|  0|  0|  0|
|GG Bridge Authority|  2|  0|  0|  0|  0|
|               SFSU|  2|  6|  0|  0|  0|
|               SFRA|  2|  0|  0|  0|  0|
..output truncated..
```

数据整理DataFrame

请注意，因为“owner”列看上去是高基数维度，因此结果被截断为前 20 行数据。虽然 Pandas 和 R 的用户应该能很好地理解 Spark DataFrame API 中的许多操作和功能，但由于 DataFrame 本质上的不可变性和分布式特性，所以它的一些有别于 Pandas/R 的地方应该被注意。例如，尽管 Spark 在加载时尽可能推断数据类型，但默认的回退类型（fallback type）还是字符串，这一点在 SF 停车示例中的“regcap”列有所体现。在 Pandas 中，可以通过选择该列并使用 `astype` 轻松转换该列值的类型：

```
parking['regcap'].astype(int)
```

但由于 DataFrame 实际上只是 RDD 的封装，是不可变的集合，因此需要执行几个步骤才能将此列转换为 `int` 类型。这种解决方法会根据现有列创建一个新列，将其值转换为正确的类型，最后删除旧列。为了保留列名，首先使用 `withColumnRenamed` 方法将现有列重命名为“regcap_old”，然后使用 `withColumn` 方法添加新的“regcap”列，该列包含 `regcap_old` 中转换类型后的值：[6]

```
parking = parking.withColumnRenamed('regcap', 'regcap_old')
parking = parking.withColumn('regcap', parking['regcap_old'].cast('int'))
parking = parking.drop('regcap_old')
parking.printSchema()
```

因为其他数值列也需要进行这个转化，所以本着 DRY 的精神，来定义一个工具函数，为任意列和数据类型执行这种转换：

```
def convert_column(df, col, new_type):
    old_col = '%s_old' % col
    df = df.withColumnRenamed(col, old_col)
    df = df.withColumn(col, df[old_col].cast(new_type))
    df.drop(old_col)
    return df

parking = convert_column(parking, 'valetcap', 'int')
parking = convert_column(parking, 'mccap', 'int')
parking.printSchema()
```

注 6：虽然这里使用的是 Spark 的 `cast` 方法，但是从 Spark 1.4 起也可以使用 `astype`，它是 `cast` 方法的 Pandas 友好别名。

不幸的是，这个函数不能处理“latitude”和“longitude”，因为它们实际上是“location_1”结构中的字段。我们可以进行一些改良，定义另一个参数为“location_1”结构类型的函数，使用 Google 的 Geocoding API（http://bit.ly/1r2xH5F）执行经纬度查找，以返回邻域名称。使用 requests（http://docs.python-requests.org/en/master/）库来发送请求：

```
import requests

def to_neighborhood(location):
    """
    使用Google的Geocoding API执行经纬度的逆向查找
    https://developers.google.com/maps/documentation/geocoding/intro#reverse-
example
    """
    name = 'N/A'
    lat = location.latitude
    long = location.longitude

    r = requests.get(
    'https://maps.googleapis.com/maps/api/geocode/json?latlng=%s,%s' %
    (lat, long))

    if r.status_code == 200:
        content = r.json()
        # results是匹配地址的列表
        places = content['results']
        neighborhoods = [p['formatted_address'] for p in places if
        'neighborhood' in p['types']]

        if neighborhoods:
            # 地址格式为Japantown, San Francisco, CA
            # 所以根据逗号分割,返回邻域名称
            name = neighborhoods[0].split(',')[0]

    return name
```

to_neighborhood 函数接受一个 location 结构并返回一个字符串类型，但是如何在列表达式中使用这个函数呢？ pyspark.sql.functions 模块提供了用于注册 UDF 的 udf 函数。通过向 UDF 传递一个可调用的 Python 函数和该函数的返回类型对应的 Spark SQL 数据类型，我们声明了一个内联 UDF；在这个示例中，返回的是一个字符串，所以使用 pyspark.sql.types 中的 StringType 数据类型。注册后，可以使用该 UDF 通过一个 withColumn 表达式重新格式化“location_1“列：

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

location_to_neighborhood=udf(to_neighborhood, StringType())

sfmta_parking = parking.filter(parking.owner == 'SFMTA') \
                        .select("location_1", "primetype", "landusetyp",
                        "garorlot", "regcap", "valetcap", "mccap") \
                        .withColumn("location_1",
                        location_to_neighborhood("location_1")) \
                        .sort("regcap", ascending=False)
```

```
sfmta_parking.show()

+-----------------+---------+----------+--------+------+--------+-----+
|       location_1|primetype|landusetyp|garorlot|regcap|valetcap|mccap|
+-----------------+---------+----------+--------+------+--------+-----+
|  South of Market|      PPA|          |       G|  2585|       0|   47|
|              N/A|      PPA|          |       G|  1865|       0|    0|
|Financial District|      PPA|          |       G|  1095|       0|    0|
|     Union Square|      PPA|          |       G|   985|       0|    0|
.. output truncated ..
```

在 Spark 本地模式下，由于 Python 的**全局解释器锁**（global interpreter lock，GIL，https://wiki.python.org/moin/GlobalInterpreterLock）的线程限制，我们无法并行化对 API 的 HTTP 请求。因此，在本地模式下使用此实用程序将需要串行运行整个 RDD，这可能需要相当长的时间。因此，为了让操作在合理的时间内完成，此示例将 DataFrame 过滤到了合适的大小。

如你所见，使用 Spark 的 DataFrame API 定义、注册 UDF 的过程比使用 Pig 和 Hive 容易得多。一旦注册，UDF 不仅可以被同一个 Spark 集群上的其他程序使用，也可以被通过 JDBC/ODBC 接口连接到 Spark SQL 上的 BI 工具使用。这使得 `udf` 函数轻松成为了 DataFrame API 提供的最强大的函数，因为它向 SQL 用户展现了应用高级计算或操作的无限可能性。本书没有涉及的内置功能和函数还有很多，它们的数量也会随着 Spark 版本的发布而不断增长。

要查看受 pyspark 的 Spark SQL 和 DataFrame API 支持的类和函数的最新列表，请参见官方 API 文档。Spark 开发新闻的另一个重要来源是 Databricks 公司（https://databricks.com/），由 Spark 创始人创立。Databricks 经常发布博文，描述所有新添加到 API 中的主要功能，例如“Statistical and Mathematical Functions with DataFrame in Spark”（http://bit.ly/26B8HDd）。

8.3　小结

在本章中，我们了解了 Pig 如何大大简化了 MapReduce 数据流水线的构建过程。Pig 的主要使用场景是传统的 ETL 数据流水线过程，但它也是一种很好的工具，非常适用于执行临时分析，并从大批量数据中构建迭代处理或预测模型，当分析越来越复杂时尤其如此。

我们还介绍了 Spark SQL 模块和 DataFrame API。Spark 提供了内置集成，支持对结构化数据集的关系处理，并允许用户在单个编程环境中将关系处理和复杂的分析相结合。DataFrame 为 Hadoop 或 Spark 的 Python 程序员提供了广泛的、前所未有的分析可能性。推荐你阅读你偏爱的语言栈的 Spark 官方 DataFrame API 文档（http://spark.apache.org/docs/latest/sql-programming-guide.html），进一步探索 Spark 的 DataFrame API；并时刻关注 Spark 新闻（http://spark.apache.org/news/），留意未来发展。

第 9 章

机器学习

机器学习计算旨在从当前和历史数据中推导出预测模型。它作出了一个固有假设，即经历越多训练或获取越多经验，学习获得的算法将改进越多。通过从大数据集训练出来的模型，机器学习算法可以在非常小的领域实现非常好的预测效果。

因此，大多数机器学习算法都涉及大规模计算。出于这个原因，机器学习计算非常适用于 Spark 等分布式计算范式，利用大型训练集生成有意义的结果。本章将介绍 Spark 内置的机器学习库——Spark MLlib（http://spark.apache.org/docs/1.5.0/mllib-guide.html）。它由许多常见的学习算法和实用程序组成，比如分类、回归、聚类、协同过滤、降维以及一个新的“机器学习流水线”框架——spark.ml。spark.ml 提供了一套统一的高级 API，可以帮助用户创建和优化实际的机器学习流水线。[1]

9.1　使用Spark进行可扩展的机器学习

在第 4 章中，我们将 Spark 作为一个可在 Hadoop 集群上运行的内存分布式计算引擎进行了介绍。而且，Spark 平台还附带了几个使用 Spark 处理引擎的内置组件，来支持其他类型的分析工作，这些功能都受益于 Spark 的计算优化。本章将仔细研究 Spark 的内置机器学习库——MLlib。该库包含一套通用的统计和机器学习算法和实用程序，它们都被设计为能在集群中扩展。[2]

有些人可能对数据挖掘和机器学习的编程库很熟悉，比如 Python 的 Weka（http://www.cs.waikato.ac.nz/ml/weka/）或者 Scikit-Learn（http://scikit-learn.org）。虽然用这些库能游刃

注 1：参见 Spark 的 Machine Learning Library (MLlib) Guide（http://spark.apache.org/docs/1.5.0/mllib-guide.html）。

注 2：《Spark 快速大数据分析》，Holden Karau 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。

有余地应对可以在单个机器上处理的中小型数据集，但对于要求分布式存储和并行处理的大型数据集，我们不仅需要可以处理分布式数据集的计算引擎，还需要为并行平台设计的算法。Spark MLlib 仅仅包含并行算法，使用 Spark 的 RDD 操作跨节点并行应用操作。好在有许多机器学习技术和算法都非常适合并行化。但一定要记住，与 Spark API 一样，使用 Spark MLlib 时要注意创建数据（作为 RDD），并以分布式并行化的方式对数据进行操作。例如，对一个原始类型的小数据集（Python 字典或列表）调用 `parallelize()`，以便将其提供给集群中的所有节点。

Spark MLlib 包括一些统计和机器学习技术，比如采样、相关计算、假设检验等。而我们将主要关注 MLlib 的机器学习算法。这类算法基于训练数据寻找算法行为的数学最优解，从而作出预测和决策。[3]Spark MLlib 学习算法集中在机器学习的三个关键领域，通常被称为机器学习的 3C。

协同过滤（collaborative filtering）
: 也被称为推荐引擎。它基于过去的行为、偏好或与已知实体 / 用户的相似性产生推荐。

分类（classification）
: 也被称为有监督学习。它从监督训练集中学习，并根据该训练集对未分类项进行分类。

聚类（clustering）
: 也被称为无监督学习。它基于类似特征，将数据分组为集群。

一般来说，要想实现这些算法，得先从数据中定义和提取一组特征作为特征的数值表示。例如，如果我们要设计一个能推荐具有相似属性（价格、颜色、品牌等）产品的推荐系统，就可以定义由每个产品属性的加权数值组成的特征向量。或者，当我们想提取非结构化文本的特征（即基于垃圾邮件，检测过滤电子邮件）时，则可以用每个词在每个分类类别（即是垃圾邮件或不是垃圾邮件）的**词频－逆文档频率**（TF-IDF）的向量来表示它。

一旦从数据中提取了特征向量，就可以将它们作为训练数据提供给机器学习算法，该算法将返回表示预测的训练模型。在训练有监督学习模型时，通常会保留一部分训练数据作为"测试数据"，将模型应用于测试数据，并通过比较测试数据的预测结果与实际结果来量化模型的准确性。这使我们能评估模型的准确性并优化其精度。机器学习流水线的概览图如图 9-1 所示。

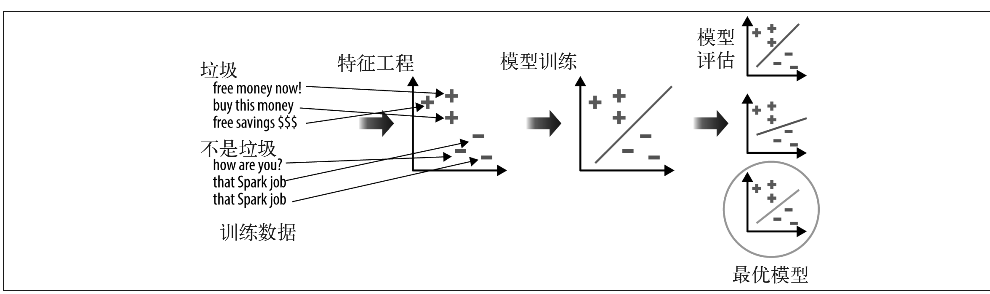


图 9-1：机器学习流水线

注 3：SNN Adaptive Intelligence, "What Is Machine Learning?" (http://www.mlplatform.nl/what-is-machine-learning/).

我们将把这种常见的机器学习流水线应用于接下来的几个例子中，并使用 MLlib 内置的一些评估工具来评估我们的学习模型。假设你已经安装了 Spark 并符合运行 Spark MLlib 的要求，具体内容请参见附录 B。

9.1.1 协同过滤

协同过滤（或**推荐系统**）应该最常见于电子商务领域，比如亚马逊和 Netflix 等公司通过挖掘用户行为数据（如浏览、评分、点击和购买）来推荐其他产品。 广义上，协同过滤算法分为两种。

基于用户的推荐系统

查找与目标用户相似的用户，使用其协同评分为目标用户提供推荐。

基于物品的推荐系统

查找并推荐与目标用户相关联物品相似或相关的物品。

MLlib 的协同过滤库专注于基于用户的推荐，使用交替最小二乘法（alternating least squares，ALS）算法的实现。[4]MLlib 的协同过滤方法将用户的偏好表示为用户物品关联矩阵，其中每位用户和物品的点积通过将偏好评分（或评级）与加权因子相乘获取。这样，我们就能接收用户的显式反馈（例如正评分、购买）和隐式反馈（例如浏览、点击），并将它们并入模型中，作为二元偏好和置信度值的组合。然后，该模型将去寻找可用于预测物品的预期偏好的隐语义因子。

示例：一个基于用户的推荐系统

试着用 MLlib 的 ALS 算法为在线约会服务生成推荐（或潜在）对象。我们将根据已有的交友网站的个人资料评分数据集为特定用户生成推荐内容。

在 GitHub 仓库的 data/mllib/dating 目录中，你可以找到包含数据集的两个 CSV 文件：168 791 份用户个人资料（gender.dat），以及一百万条以上关于他们的用户评分数据（ratings.dat）。这些数据可从 Occam 的实验室（http://www.occamslab.com/petricek/data）获取。

评分数据遵从以下格式：UserID、ProfileID、Rating。UserID 是提供评分的用户，ProfileID 是被评分的用户，Rating 是 1~10 的评分，其中 10 是最高分。

UserID 的范围介于 1~135 359，ProfileID 的范围介于 1~220 970（并非每份用户资料都被评过分）。只有至少提供了 20 条评分的用户才会被纳入，一直打同样分数的用户将被排除。

用户的性别信息遵从以下格式：UserID、Gender，其中男性为“M”、女性为“F”、未知为“U”。

可运行的完整约会推荐程序可以在 GitHub 仓库的如下目录中找到：

```
hadoop-fundamentals/mllib/collaborative_filtering/als/matchmaker.py
```

使用 `spark-submit` 命令，并向它传递两个参数：UserID（要为谁生成推荐）以及 M 或 F

注 4：Koren Yehuda et al.,“Matrix Factorization Techniques For Recommender Systems”(http://dl.acm.org/citation.cfm?id=1608614), Computer 14.8(2009):30–37.

（对伴侣的性别偏好），该程序就可以在 Spark 上运行了。建议将此输出通过管道写入到一个文件中：

```
$ $SPARK_HOME/bin/spark-submit \
~/hadoop-fundamentals/mllib/collaborative_filtering/als/matchmaker.py 1 M \
> ~/matchmaking_recs.txt
```

我们将分析该程序的每个主要步骤。首先，使用应用程序的名称配置 SparkContext，并将每个 executor 使用的内存大小设置为 2GB，因为 ALS 算法处理这个数据量需要大量内存：

```
# 配置Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("Dating Recommender") \
                  .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

接下来，读取 UserID 参数以及用户的性取向，并对评分文件中的每条记录调用自定义的 `parse_rating` 方法：

```
def parse_rating(line, sep=','):
    """
    解析评分行
    返回:元组(随机整数, (user_id, profile_id, rating))
    """
    fields = line.strip().split(sep)
    user_id = int(fields[0]) # 将user_id转换为int
    profile_id = int(fields[1]) # 将profile_id转换为int
    rating = float(fields[2]) # 将rated_id转换为int
    return random.randint(1, 10), (user_id, profile_id, rating)
```

给定一个评分行，`parse_rating` 方法返回一个元组，其中第一项是一个随机整数，第二项是另一个元组 `(user_id, profile_id, rating)`：

```
matchseeker = int(sys.argv[1])
gender_filter = sys.argv[2]

# 创建评分RDD (随机整数, (user_id, profile_id, rating))
ratings = sc.textFile(
    "/home/hadoop/hadoop-fundamentals/data/dating/ratings.dat")\
    .map(parse_rating)
```

为元组中的第一项生成一个随机数，之后它将作为键将此 RDD 分解为训练集和测试集。ALS 要求将 `Rating` 对象表示为 `(UserId, ItemId, Rating)` 元组。在这个示例中，`ItemId` 实际将映射到其他用户资料的用户 ID。

接下来，通过将自定义的 `parse_user` 方法映射到 gender.dat 的每一行，读取用户个人资料数据：

```
def parse_user(line, sep=','):
    """
    解析用户行
    返回:元组(user_id, gender)
    """
    fields = line.strip().split(sep)
```

```
        user_id = int(fields[0]) # 将user_id转换为int
        gender = fields[1]
        return user_id, gender
```

给定一个用户行，parse_user 方法返回一个元组（user_id，gender）。一旦用户元组的 RDD 生成，就调用 collect() 将 RDD 转换为列表：

```
# 创建用户RDD
users = dict(sc.textFile(
"/home/hadoop/hadoop-fundamentals/data/dating/gender.dat")\
.map(parse_user).collect())
```

现在来将评分数据分为训练集和验证集，训练集用于训练模型，验证集用于评估模型。通过对添加到每个元组的随机整数键进行过滤，保留 60% 的训练数据和 40% 的验证数据。将分区数设置为 4（或计算机支持的处理器内核数）并缓存结果，提高 RDD 的并行性。

```
# 基于时间戳的最后一位,创建训练集(60%)和验证集(40%)
num_partitions = 4
training = ratings.filter(lambda x: x[0] < 6) \
                  .values() \
                  .repartition(num_partitions) \
                  .cache()

validation = ratings.filter(lambda x: x[0] >= 6) \
                    .values() \
                    .repartition(num_partitions) \
                    .cache()

num_training = training.count()
num_validation = validation.count()

print "Training: %d and validation: %d\n" % (num_training, num_validation)
```

可以通过设置和调整 ALS 提供的这些训练参数来优化模型。

rank
: 使用的特征向量的大小，由隐语义因子的数量决定；rank 越大，产生的模型越好，但计算的代价也更大（默认值为 10）。

num_iterations
: 迭代次数（默认值为 10）。

lambda
: 正则化参数（默认值为 0.01）。

alpha
: 常数（默认值为 1.0），用于计算隐式反馈 ALS 的置信度。

因为此例只捕获显式评分，所以将忽略 alpha 并使用默认值。其他参数中，rank 使用 8，将迭代次数设置为 8，lambda 为 0.1。由于对数据还不够了解，无法确定隐语义因子的数量或合适的正则化值，所以这些初始训练参数设置得有些随意。然而，可以先从这个组合开

始，将它与使用其他训练参数组合的模型进行对比来评估结果，确定最佳拟合模型：

```
# rank是模型中隐语义因子的数量
# num_iterations是迭代次数
# lambda指定ALS中的正则化参数
rank = 8
num_iterations = 8
lambda = 0.1
```

现在使用 `ALS.train()` 方法创建模型，该方法接受评分元组的训练 RDD 和我们的训练参数：

```
# 使用训练数据、已配置的rank和迭代参数训练模型
model = ALS.train(training, rank, num_iterations, lambda)

# 使用验证集评估经过训练的模型
print "The model was trained with rank = %d, lambda = %.1f, and %d iterations.
\n" % \
    (rank, lambda, num_iterations)
```

在详细日志记录模式下运行 `train()` 方法时要小心，此操作需要进行几次 RDD 投影和操作，因此可能会有长达几分钟的日志滚动。

模型一旦被创建，就使用均方根误差（root mean squared error，RMSE）来计算每个模型的误差。RMSE 是拥有实际评分的所有用户的（**实际评分 – 预测评级）^2 的平均值的平方根**。[5]

$$RMS = \left(\frac{1}{n}\sum_{i=1}^{n}\left(\text{model}_i - \text{observed}_i\right)^2\right)^{\frac{1}{2}}$$

我们的推荐程序也可以相应地实现 RMSE 计算：

```
def compute_rmse(model, data, n):
    """
    计算RMSE,或者(实际评分–预测评分)^2的平均值的平方根
    """
    predictions = model.predictAll(data.map(lambda x: (x[0], x[1])))
    predictions_ratings = predictions.map(lambda x: ((x[0], x[1]), x[2])) \
      .join(data.map(lambda x: ((x[0], x[1]), x[2]))) \
      .values()
    return sqrt(predictions_ratings.map(lambda x: (x[0] - x[1]) ** 2). \
    reduce(add) / float(n))
```

RMSE 表示模型对数据的绝对拟合（观察数据点与模型预测值的接近程度），并且与评分值的单位相同。RMSE 值越小，拟合度越高。但由于它与评分值相关，所以应该按 1~10 进行评估。根据结果，可以通过调整训练参数，或者提供更多或更好的训练数据，来优化模型：

注 5：Kaggle,“Root Mean Squared Error”(https://www.kaggle.com/wiki/RootMeanSquaredError).

```
# 打印模型的RMSE
validation_rmse = compute_rmse(model, validation, num_validation)

print "The model was trained with rank=%d, lambda=%.1f, and %d iterations." % \
(rank, lambda, num_iterations)
print "Its RMSE on the validation set is %f.\n" % validation_rmse
```

假设我们对 RMSE 值反映出来的模型拟合程度很满意，就可以用它来为给定用户生成推荐了。通过根据给定用户的性取向进行过滤，先生成一组符合条件的用户。这是推荐候选人 RDD：

```
# 根据性取向进行过滤
partners = sc.parallelize([u[0] for u in filter(lambda u: u[1] ==
 gender_filter, users.items())])
```

现在使用模型的 `predictAll()` 方法，向它传递一个二元元组 RDD，RDD 的 key 是 `user_id`，其中 `user_id` 是给定的要为之生成推荐的用户（matchseeker）。这样，模型就能生成推荐。我们会将结果收集到一个列表中，按照评分值降序排序，取前 10 名推荐用户：

```
# 使用经过训练的模型进行预测
predictions = model.predictAll(partners.map(lambda x: (matchseeker, x))) \
.collect()

# 对推荐进行排序
recommendations = sorted(predictions, key=lambda x: x[2], reverse=True)[:10]
```

最后，打印完整的推荐列表并停止 SparkContext：

```
print "Eligible partners recommended for User ID: %d" % matchseeker
for i in xrange(len(recommendations)):
    print ("%2d: %s" % (i + 1, recommendations[i][1])).encode('ascii', 'ignore')

# 清理
sc.stop()
```

如果使用前面的命令将此作业提交到 Spark，并将输出保存到一个结果文件中，应该会看到类似于以下内容的输出：

```
$ cat matchmaking_recs.txt

Training: 542953 and validation: 542279

The model was trained with rank = 8, lambda = 0.1, and 8 iterations.

Its RMSE on the validation set is 3.580347.

Eligible partners recommended for User ID: 1
 1: 100939
 2: 70020
 3: 109013
 4: 54998
 5: 132170
 6: 3843
 7: 170778
```

```
 8: 51378
 9: 8849
10: 118595
```

RMSE 是评估模型性能的重要指标。与大多数机器学习算法一样，协同过滤模型中的数据和迭代次数越多，模型表现越好。建议你尝试对 ALS 使用不同的 `rank`、迭代次数和正则化（`lambda`）参数的组合，比较 RMSE 以找到最佳拟合的参数组合。你可以在 Spark MLlib 文档的“Collaborative Filtering”（http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html#scaling-of-the-regularization-parameter）找到更多有关参数调优的信息，以及一个实现电影推荐系统的 ALS 算法示例。

9.1.2　分类

分类试图通过有监督训练方法将数据（通常是文本或文档）分类，这些方法使用带标注的训练集来发现模式，使机器学习程序能快速标记新记录。例如，一个简单的分类算法可能会记录与类别相关联的特征和词，以及该词出现在给定类别中的次数。一旦机器学习程序从训练数据中提取出了特征，它就可以生成特征向量并应用统计模型构建预测模型，然后将预测模型应用于新的数据。

MLlib 提供了一些用于二分类、多分类以及回归分析的算法。在二分类中，我们希望将实体分为两个不同的类别或标签（例如确定电子邮件是否为垃圾邮件）；在多分类中，我们希望将实体分类为两个以上的类别（例如确定新闻报道属于哪个类别）；回归分析算法的目标是，使用连续函数估计因变量（例如身体活动水平）与一个或多个自变量（例如心脏病的风险）之间的关系和依赖性。

在这些类型的算法中，MLlib 实现都要对一组**带标签**的例子（example）应用算法。这些例子被表示为 `LabeledPoint` 对象，包括一个数值（用于二分类）或特征向量（用于多分类）以及类别标签。已经分类的 `LabeledPoints` 中的训练数据用于训练模型，然后用模型去预测新实体的类别。

MLlib 的官方文档（http://bit.ly/26Bp4zE）按类别列举了所有支持的分类算法。本节将通过随机梯度下降的逻辑回归过程（也被称为 `LogisticRegressionWithSGD`）创建一个简单的二分类器。

示例：一个逻辑回归分类

在这个示例中，我们将构建一个简单的垃圾邮件分类器，使用已经分类（垃圾邮件和非垃圾邮件）的电子邮件数据进行训练。此垃圾邮件分类器将使用两个 MLlib 算法：`HashingTF` 和 `LogisticRegressionWithSGD`，前者从训练文本提取词频向量作为特征向量，后者使用随机梯度下降（http://bit.ly/26Bp7vf）实现逻辑回归。

可以在 GitHub 仓库的 /data 目录中的 spam_classifier.zip 中找到训练数据 spam.txt 和 ham.txt。此数据是 SpamAssassin 公共语料库（http://spamassassin.apache.org/publiccorpus/）的一个子集。完整的垃圾邮件分类程序可以在 mllib/classification 目录下找到，可以使用以下命令运行程序：

```
$ $SPARK_HOME/bin/spark-submit \
/home/hadoop/hadoop-fundamentals/mllib/classification/spam_classifier.py \
/home/hadoop/hadoop-fundamentals/data/spam_classifier/spam.txt \
/home/hadoop/hadoop-fundamentals/data/spam_classifier/ham.txt
```

我们将讲解每个主要步骤。

首先配置 SparkContext，设置应用程序名称，并将 executor 内存增加到 2GB：

```
# 配置Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("Spam Classifier") \
                  .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

接下来读取命令行参数，获取训练数据文件的路径。读取这些文件，创建垃圾邮件（spam）和非垃圾邮件（ham）RDD：

```
spam_file = sys.argv[1]
ham_file = sys.argv[2]

spam = sc.textFile(spam_file)
ham = sc.textFile(ham_file)
```

现在，实例化 `HashingTF` 对象，将要提取的特征数量设置为 10 000：

```
tf = HashingTF(numFeatures=10000)
```

将 `HashingTF` 的 `transform()` 方法应用于 spam 和 ham 数据，首先将内容分成单词令牌。这将从 spam 和 ham RDD 中提取出词频向量，并将其投影为新的特征向量 RDD：

```
spam_features = spam.map(lambda email: tf.transform(email. \
 split(" ")))
ham_features = ham.map(lambda email: tf.transform(email. \
 split(" ")))
```

现在将 RDD 中的每个特征向量转换为 `LabeledPoint`。因为这是一个二分类器，因此用 1 表示垃圾邮件，用 0 表示非垃圾邮件。`LabeledPoint` 对象的第二个值将包含该特征。将这些 RDD 的并集作为训练数据集并缓存它，因为逻辑回归是一种迭代算法：

```
positive_examples = spam_features.map(lambda features: LabeledPoint(1, features))
negative_examples = ham_features.map(lambda features: LabeledPoint(0, features))
training = positive_examples.union(negative_examples)
training.cache()
```

现在使用 SGD 算法和训练数据运行逻辑回归：

```
model = LogisticRegressionWithSGD.train(training)
```

现在创建测试数据，包括应分类为垃圾邮件的文本内容，以及应分类为非垃圾邮件的文本内容。然后使用训练模型来预测测试数据是否被视为垃圾邮件。回想之前对 `LabeledPoints` 的设置，1 表示垃圾邮件，0 表示非垃圾邮件：

```
# 创建测试数据,测试模型
positive_test = tf.transform("Guaranteed to Lose 20 lbs in 10 days
```

```
Try FREE!".split(" "))
negative_test = tf.transform("Hi, Mom, I'm learning all about Hadoop
and Spark!".split(" "))

print "Prediction for positive test example: %g" % model.predict(positive_test)
print "Prediction for negative test example: %g" % model.predict(negative_test)
```

至此，就可以将预测结果与数据的分类进行比较，评估分类器模型的准确性，或者将该模型应用于未标记的数据集。MLlib 的分类算法针对大型监督训练数据进行了优化。因此，比起少而精，更多的数据通常会产生更好的效果。然而，分析数据并应用最合适的算法和评估方法仍然很重要。请参考官方的 Spark MLlib 文档（http://spark.apache.org/docs/latest/ml-guide.html），了解所有支持的分类算法和评估指标，特别要注意它们各自支持的 API（Scala、Java、Python）。

9.1.3 聚类

与协同过滤和分类算法不同，聚类利用无监督学习技术来构建模型。聚类算法尝试将数据集合组织成类似项目的分组，比如寻找具有相似特征或兴趣的客户群体，或将动植物按常见物种分组。聚类的目标是将数据分成多个簇，使每个簇内的数据彼此之间比与其他簇中的数据更相似。[6]

Spark MLlib 提供了一些流行的聚类模型，但其中最简单也是最流行的聚类算法恐怕还得属 *k*-means。*k*-means 算法需要将所有对象表示为一组数值特征，并事先指定想要的目标簇数（*k* 个簇）。

MLlib 的 *k*-means 聚类的实现也从向量化数据集开始，将每个对象表示为 *n* 维空间中的特征向量，其中 *n* 用于描述要聚类的对象的所有特征的数量。算法首先在该向量空间随机选择 *k* 个点作为簇的初始中心或**质心**，然后将每个对象分配给最接近的质心，使用簇中所有点的坐标的平均值重新计算质心点，并根据需要将对象重新分配到最近的簇。分配对象和重新计算中心的过程不断重复，直到过程收敛，如图 9-2 所示。[7]

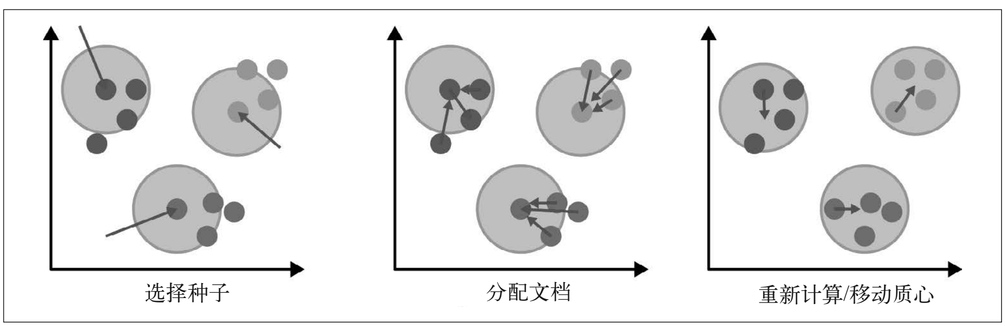


图 9-2：*k*-means 聚类算法的计算阶段

注 6：《Hadoop 硬实战》，Alex Holmes 著。

注 7：《Spark 快速大数据分析》Holden Karau 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。

聚类中最重要的问题就是确定如何量化要聚类的对象的相似度。加权方法可以从 TF-IDF 获得，这对于文本文档特别有用；另一种加权方法是通过数据中的其他自定义属性（使用计算指标衡量的平均占比）的函数来确定（例如基于以美元为单位的总购买金额划分顾客）。对于 MLlib 的 *k*-means 聚类的输入，需要指出对特征向量使用的加权方法。例如，如果确定要根据总购买金额、平均购买频率和每次平均购买金额这三个特征对所有客户进行聚类，那么客户样本可能如表 9-1 所示。

表9-1：客户特征向量化

姓名	总购买金额（美元）	每月平均购买次数	每次平均购买金额	特征向量
Jane	825	5	115	`[825,5,115]`
Bob	201	1	45	`[201,1,45]`
Emma	649	2	65	`[649,2,65]`

有多个特征时一定要注意，维度值是以不同单位表示的，或者尚未进行归一化。如果使用简单的、基于距离的指标来确定这些向量之间的相似性，总购买金额将主导结果。通过给不同的维度加权，可以解决这个问题。[8]

示例：一个*k*-means聚类

在这个例子中，我们将应用 *k*-means 聚类算法来确定截至今年美国哪些区域发生地震的次数最多。[9] 这些信息可以在 GitHub 仓库的 /data 目录中的 earthquakes.csv 文件中找到。此 CSV 文件的列如下所示：

- time
- latitude
- longitude
- depth
- magnitude
- magnitudeType
- nst
- gap
- dmin
- rms
- net
- jd
- updated
- place

从这些记录中提取纬度（latitude）和经度（longitude），并用其训练模型。在这个迭代过程中，我们将尝试生成 6 个簇。完整的程序可以使用以下命令运行：

注 8：《Mahout 实战》，Sean Owen、Robin Anil、Ted Dunning、Ellen Friedman 合著。

注 9：参见“USGS Earthquakes Hazard Program”（http://earthquake.usgs.gov/earthquakes/feed/v1.0/csv.php）。

```
$ $SPARK_HOME/bin/spark-submit \
/home/hadoop/hadoop-fundamentals/mllib/clustering/earthquakes_clustering.py \
/home/hadoop/hadoop-fundamentals/data/earthquakes.csv \
6 > clusters.txt
```

首先，配置 Spark 并创建 SparkContext：

```
# 配置Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("Earthquake Clustering") \
                  .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

接下来，从地震数据文件创建训练 RDD，解析每一行的纬度和经度，并将其转化为一个 NumPy 数组：

```
# 创建用于训练的(lat, long) RDD向量
earthquakes_file = sys.argv[1]
training = sc.textFile(earthquakes_file).map(parse_vector)
```

使用第二个参数设置 *k* 个簇，在本示例中为 6：

```
k = int(sys.argv[2])
```

调用 `KMeans.train()`，将训练集和 *k*（设置为 6）传递给它。这将生成模型，我们也可以访问簇的中心：

```
# 基于训练数据和k-clusters训练模型
model = KMeans.train(training, k)

print "Earthquake cluster centers: " + str(model.clusterCenters)
sc.stop()
```

如果检查输出的 clusters.txt 文件，就会看到类似如下的输出：

```
Earthquake cluster centers: [array([ 38.63343185, -119.22434212]),
array([ 13.9684592 , 142.97677391]),
array([ 61.00245376, -152.27632577]),
array([ 35.74366346, 27.33590769]),
array([ 10.8458037, -158.656725 ]),
array([ 23.48432962, -82.3864285 ])]
```

现在，就可以根据训练数据绘制结果输出，对结果执行“眼球”评估，并通过优化簇数（*k*）和迭代次数来调整簇中心了。为了获取更精确的评估指标，还可以计算“集内平方误差的总和”（Within Set Sum of Squared Errors），它衡量每个中心点周围簇点的紧凑度：

```
def error(point):
    center = model.centers[model.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = training.map(lambda point: error(point)).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))
```

9.2 小结

本章实现了一个基于用户的简单推荐系统，使用实现逻辑回归的二分类器对电子邮件进行了分类，使用 *k*-means 聚类算法对文档集合进行了聚类，并介绍了一点输入数据的向量表示。但是，这只是 MLlib 的预测分析能力的皮毛。

除了其他算法和数据准备工具，MLlib 还提供了评估算法的质量和性能的工具。希望这个简短的介绍展示了 MLlib 将强大的统计学习技术应用于大型数据集的潜力。由于 Spark MLlib 正逐步发展成一个更广泛的分布式机器学习框架，因此我们也鼓励你去深入了解它的统计和机器学习能力以及未来发展情况。数据类型、算法和实用程序都可以在官方的 Spark MLlib 指南（http://spark.apache.org/docs/latest/mllib-guide.html）中找到。我们还推荐你阅读 Sandy Ryza、Uri Laserson、Sean Owen 和 Josh Wills 合著的《Spark 高级数据分析》[10]，这是一本以示例驱动的优秀图书。

注 10：本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1668。

第 10 章

总结：分布式数据科学实战

在本书中，我们领略了 Hadoop 生态系统的具体组成部分。第一部分讨论了如何与集群进行交互以及如何使用集群。如前所述，Hadoop 是一种分布式计算的操作系统；和本地计算机上提供文件系统和进程管理的操作系统一样，Hadoop 通过 HDFS 以及 YARN 形式的资源和调度框架提供分布式数据存储和访问。HDFS 和 YARN 一起构成了一种在极大数据集上进行分布式分析的机制。

编写分布式作业的原始方法是使用 MapReduce 框架，它让你能指定 mapper 和 reducer 任务。将这些任务链接在一起，可以进行更庞大的计算。由于 Python 是数据科学最流行的工具之一，所以本书专门探讨了如何通过 Hadoop Streaming 执行使用 Python 脚本的 MapReduce 作业。在第一部分中，还探索了一种更纯粹的解决方案：使用 Spark 的 Python API 在使用了 YARN 的 Hadoop 集群中执行 Spark 作业。最后，介绍了在集群上常用的分布式分析和设计模式，结束了对低级工具的讨论。

第二部分从低级编程细节完全转移到了数据挖掘、数据采集、数据流和机器学习所用的高级工具上。这一部分主要针对通过 Hadoop 的各种现有工具进行分布式数据分析的日常情形，以及如何根据大数据流水线（数据采集、数据整理 /staging、计算和分析、工作流管理）组织这些工具。

你可能会有一个疑问：如何将 Hadoop 和 Spark 中的所有这些工具、组件组合在一起？

在第 1 章中，我们讨论了为什么大数据变得重要了——主要由于**数据产品**（从数据中获取价值，并通过应用预测或模式识别来生成新数据的应用程序）的兴起。数据产品必须具有自适应性和广泛适用性（可通用）；因此，机器学习和强化技术在成功部署数据产品方面越来越突出。数据产品的自适应行为要求它们不能是静态的，而是要不断学习；可通用性需要大量的数据参考点来拟合模型。因此，数据产品需要分布式计算来处理多样的、快速的数据——这也正是现代机器学习的特点。

数据产品是构建的消费品（不一定完全是软件），它从数据中获取价值并生成新数据。要实现该定义，必然需要应用机器学习技术。数据驱动的应用程序只是使用数据的应用程序（包括每个软件产品），例如博客、网上银行、电子商务等。即使数据驱动的应用程序从数据中获取了价值，它也不一定会生成新的数据。

本章将详细介绍如何使用本书中讨论过的所有工具来构建数据产品，并在此过程中，回答如何将分布式计算的低级操作和高级生态系统工具拟合在一起。即便本书只是 Hadoop 和分布式计算的一个入门介绍，但我们也想在总结时提供一些建议，看看接下来能做什么。希望通过将整个数据产品和机器学习生命周期进行语境化，你能更轻松地识别和了解对工作流至关重要的工具和技术。

10.1 数据产品生命周期

构建数据产品需要建立和维护活动的数据工程流水线。流水线包括采集、整理、仓储、计算和探索性分析等多个步骤，这些步骤一同构成了数据工作流管理系统。它的主要目标是建立和实施拟合的（经过训练的）模型，其核心过程包括提取、转换和加载（ETL）过程——从应用程序上下文中提取数据，将其加载到 Hadoop 中，在 Hadoop 集群中处理数据，然后将数据 ETL 回应用程序。如图 10-1 所示，可以将这个简单的流程图看作是一个活动的或者常规的生命周期。在这个周期内，通过新的数据和交互，为用户调整和使用机器学习模型。

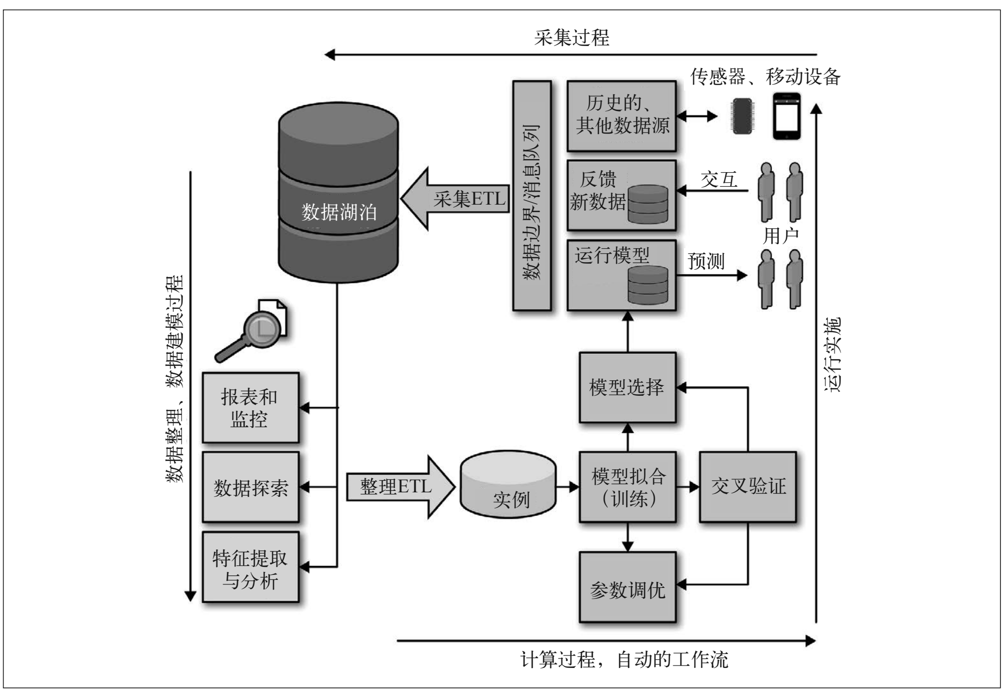


图 10-1：数据产品生命周期

如果想充分运用机器学习算法，数据产品生命周期就需要使用大数据分析和 Hadoop。拥有大量用户的应用程序必然将产生大量数据。尽管通过 128GB 内存和多个内核的强大服务器进行有效的抽样和分析后，这些数据可以被处理，但数据的**多样**和**高速**才是需要 Hadoop 和基于集群的方法的灵活性的主要原因。

灵活性确实是基于集群的系统的关键。Web 日志记录（点击流式数据）、用户交互和流式数据集（例如传感器数据）形式的输入数据源源不断地涌入应用程序。这些数据源被写入各处，比如日志文件、NoSQL 数据库，以及 Web API 后端的关系数据库。此外，来自网络爬取、数据服务和 API、调查及其他业务来源的数据等信息也不断被生成。这些额外的数据必须与现有应用程序数据被一道分析，从而确定能改进数据产品模型的特征是否存在。

因此，数据产品生命周期通常围绕一个或多个中心数据存储。数据存储极其灵活，没有约束（不像关系数据库中存在约束），但持久性强。这样的中心数据存储是 WORM 系统，即“写一次，读多次”，是向下游分析提供可靠数据的关键所在，支持历史分析和可重复的 ETL 生成（这对科学至关重要）。WORM 存储系统对数据科学非常重要，它们也因此收获了一个新名字——数据湖泊。

10.1.1　数据湖泊

传统上，我们会使用数据仓库模型来执行业务环境中的常规聚合分析。数据仓库是关系数据库的扩展，通常将数据规一化为星型模式——将多个维表连接到一个中心事实表（所以关系图看起来像星型）的模式。事务通常发生在维表上，它们的解耦使组织的各个方面在读写上有了一些性能优势。ETL 过程通过一个大连接构建“数据（超）立方”来加载事实表，我们可以在数据立方上应用枢纽分析（pivot）和其他分析方法。

为了有效利用传统的数据仓库，必须先设计一个清晰的模式，经历冗长周期的 ETL 过程，完成数据库管理、数据转换和加载后，才可以分析数据。不幸的是，当你将数据产品视为需要新数据和新数据源的、有生命的、活动的引擎时，这种传统的数据分析模型可能既费时又有限制。对应用程序的简单改动（例如新的历史数据源，或新的日志记录和提取技术）就可能改变数据立方的结构，需要重新设计星型模式的范式。这种结构调整不仅费时费力，而且还迫使我们作出一个业务决策：是否值得为这些数据增加机器，来处理新的数据量？

作为数据科学家，我们都知道所有数据至少有潜在价值，但却很难回答数据价值及其相对成本效益的问题。因此，许多公司不仅在数据仓库上有所投入，而且还开发**数据湖泊**作为主要的数据收集和同步策略。

数据湖泊支持从各种源（结构化和非结构化的）中流入未处理的原始数据，它将整个数据集合存储在一起，无须太多的组织，如图 10-2 所示。结构化数据可以从关系数据库、结构化文件（如 XML 或 JSON）和符号分隔文件（如日志文件）中获取，并且通常以基于文本的格式或某种类型的序列化二进制格式（如 SequenceFiles、Avro 或 Parquet）加载到系统。半结构化和非结构化数据包括传感器数据、二进制数据（如图像）和不是面向记录而是面向文档的文本文件（如电子邮件）。数据湖泊模式允许任何类型的数据自由流入存储，然

后通过在处理时施加所需模式的 ETL 过程流出。根据分析需求进行提取和转换后，数据被加载到一个或多个**数据仓库**进行例行分析或紧急分析。数据湖泊模式提供了在线访问整套原始的源格式数据的功能，并将模式定义延迟到处理时，让公司能在需求变化时敏捷地执行新的处理和分析。

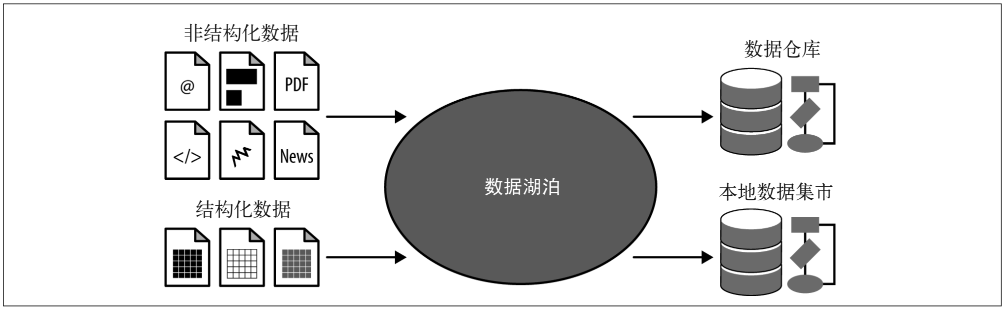


图 10-2：结构化和非结构化的数据流入数据湖泊，后者使用 ETL 过程查询，产生一个可以分析的数据仓库

尽管本书专注于 HDFS，但还有其他许多分布式数据存储解决方案，比如 GlusterFS、EMC 的 Isilon OneFS 和 Amazon 的 Simple Storage Service（S3），等等。然而，HDFS 是 Hadoop 的默认文件系统，也是构建数据湖泊非常有效的方式。HDFS 将数据分发到许多机器，支持用许多小容量的硬盘存储数据；同时，使数据可用于分布式框架中的计算，而不会产生存储区域网络（storage area network，SAN）的网络流量。此外，HDFS 会复制数据块，提供持久性和容错性，确保数据不会丢失。NameNode 以分层文件系统的形式组织数据命名空间，而不设计每个字段数据的模式。

与其使用负载过重又容量有限的单个主数据仓库，还不如将数据存储在 HDFS 数据湖泊中，通过 MapReduce 或 Spark 作业进行灵活分析，然后被提取并加载到目标系统中，例如需要特定类型分析的业务部门的企业数据仓库。此外，通常可以将在磁带上存档的、无法分析的旧历史数据转移到 Hadoop 上，进行探索性分析。如此看来，Hadoop 可以减轻传统数据仓库的沉重维护负担、突破后者在可扩展性上的限制，甚至补充了现有的数据仓库架构，如图 10-3 所示。

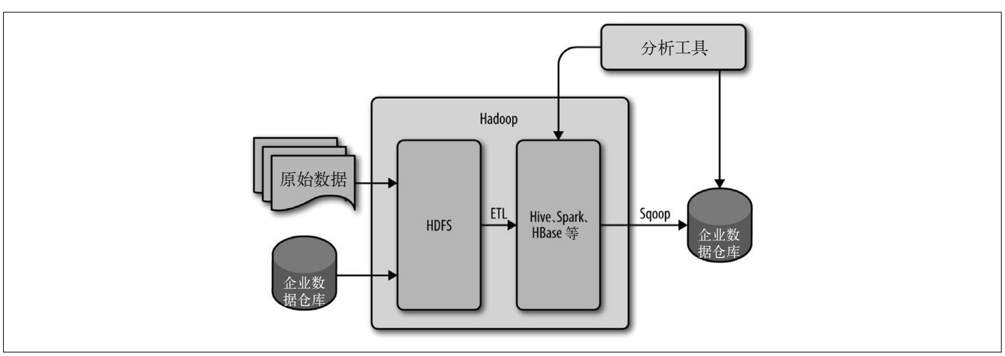


图 10-3：使用 Hadoop 的混合数据仓库架构

10.1.2　数据采集

深入了解数据产品生命周期的中心对象——数据湖泊之后，就可以将注意力转移到数据采集和数据仓储，以及数据科学家通常如何看待这些流程了。首先从数据采集开始。

一般来说，大多数数据采集从应用程序上下文获取数据，也就是和用户交互的软件产品的业务单元，或者实时收集信息的逻辑单元。例如，一个规模可观的电子商务平台可能有一个只处理客户评论的软件应用程序，和一个只收集用于安全和日志的网络流量信息的单元。这两个数据源对于异常检测（欺诈）或推荐系统等数据产品非常有价值，但必须分别采集进入数据湖泊。本节将介绍 Sqoop 和 Flume 这两种工具，它们都支持这两种上下文采集。

Sqoop 可以利用 JDBC（Java database connector）库连接到任何关系数据库系统，并将其导出到 HDFS。关系数据库几乎是目前所有 Web 应用程序和串行（非分布式）分析的后端服务器。由于关系数据库是小规模分析的着力点，在 Web 应用程序中无处不在，因此 Sqoop 是将数据从大多数大型数据源提取到 HDFS 中的重要工具。此外，由于 Sqoop 从关系上下文中提取数据，因此只要稍作整理，确保主键在数据库之间保持一致，Hive 和 SparkSQL 几乎可以马上使用从这些数据源采集的数据。在我们的示例中，Sqoop 是提取存储在关系数据库中的客户评价数据的理想工具。

另一方面，Flume 是用于采集日志记录的工具，但也可以从任何 HTTP 源中采集。Sqoop 用于结构化数据，而 Flume 主要用于非结构化数据，例如包含网络流量数据的日志。日志记录一般被认为是半结构化的，因为它们是需要解析的文本，但通常每一行都拥有相同的标准格式。Flume 还可以从 Web 请求中采集 HTML、XML、CSV 或 JSON 数据，因此能行之有效地处理特定的半结构化数据和非结构化数据的包装数据，如评论、评价或其他文本数据。因为 Flume 比 Sqoop 更通用，因此它不一定与下游数据仓储产品保持一致，并且一般要求采集过程和分析之间有 ETL 机制。

本书没有讨论消息队列**服务**。例如，Kafka 是一种分布式队列系统，可在现实世界、数据系统中的应用程序、数据湖泊三者之间创建数据边界。请求数据可以放在 Kafka 队列中，然后按需采集，而无须用户向应用程序发送请求数据，然后采集到 Hadoop 中。消息队列使得数据采集过程变得更加实时，或者至少变成了一次处理一小片，而不是像 Sqoop 那样进行大批量作业。

然而，为了获取实时数据源，还需要其他工具来处理**流式数据**。流式数据是指在线不断进入系统的无界的、可能无序的数据。Twitter 的 Storm（现在的 Heron）、MillWheel 和 Timely 等工具支持分布式的、容错的实时数据集处理。这些工具可以在 YARN 上运行，并在处理结束时将 HDFS 作为存储工具。与之类似，Spark Streaming 提供了流式数据集的微批次分析，允许以固定的间隔（例如每秒）将记录收集成一个批次，并一次性分析或使用它们。

许多现代分析架构将这些采集和处理工具组合，使其同时支持批处理和流式任务，这也被称为 **lambda 架构**，如图 10-4 所示。

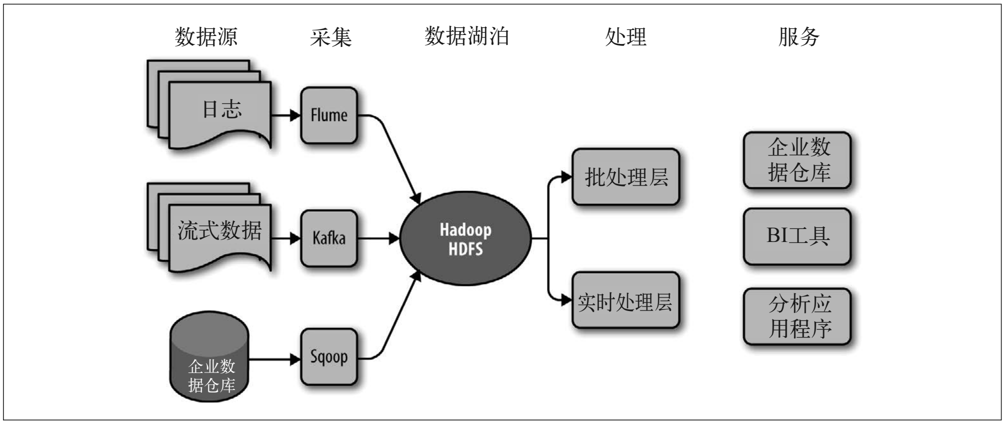


图 10-4：lambda 架构

当你将这些工具作为一个整体考虑时，可以清楚地看到，从直接将数据馈入数据仓储以进行分析的大规模批处理，到需要 ETL 和处理才能进行大规模分析的实时流中有一种连续性。至于具体采用哪种处理方式，主要由数据的具体速度和及时性（立即或在特定时间限制内完成分析）或完整性（近似与精确）共同决定。

10.1.3　计算数据存储

随着我们不断接近数据产品生命周期中更正式的仓储和分析阶段，分布式存储的需求被再一次唤醒。正如之前讨论的，通过将 Hadoop 作为数据湖泊来存储未处理的原始数据，我们可以获得相当灵活、敏捷的分析能力。然而，在很多使用场景中，结构和顺序也是必需的，在数据仓储中尤其如此——数据希望驻留在共享存储库中，维度模式为分析任务提供更简单、经过优化的查询。对于这些类型的应用程序，仅仅使用 HDFS 的文件系统接口与作为文件集合的数据进行交互是不够的；我们需要一个更高层次的接口，可以原生地理解 SQL 的结构化表语义。

1. 关系方法：Hive

在本书中，我们提出 Hive 是 Hadoop 中执行数据仓储任务的主要方法。Hive 项目包括许多组件，比如 Hive Metastore、Hive 驱动程序和执行引擎、Hive Metastore 服务和 HCatalog。其中，Hive Metastore 作为 HDFS 之上的存储管理器，存储元数据（数据库 / 表实体、列名称、类型等）；Hive 驱动程序和执行引擎将 SQL 查询编译成 MapReduce 或 Spark 作业；Hive Metastore 服务和 HCatalog 使其他 Hadoop 生态系统工具能与 Hive Metastore 进行交互。还有许多其他分布式的 SQL 或 SQL-on-Hadoop 技术来不及讨论了，它们是 Impala、Presto、Hive on Tez，等等。上述所有组件其实可以直接与 Hive Metastore 交互，或通过 HCatalog 与它交互。解决方案的选择应由数据仓储和性能要求决定，但 Hive 通常是耗时长、需要容错的查询的好选择。

在 HDFS 和 Hive 中存储数据时，一定要考虑如何以有意义且有效的方式对数据进行分区。要存储到 Hive 的话，确定分区策略时应该考虑在查询数据集时最常应用的谓词。例如，

如果要分析 `WHERE year = 2015` 或 `WHERE updated > 2016-03-15` 形式的 `WHERE` 子句，很明显，按日期过滤记录将是一个重要的访问模式，因此可以按天（例如 2016-03-01）将数据分区。这让 Hive 能只读取所需的特定分区，从而减少 I/O 量并显著缩短查询时间。[1]

不幸的是，大多数 SQL 查询都非常复杂，针对分析使用的各种谓词可能会导致大量不同的分区。这要么导致数据极度碎片化，要么会降低数据存储的灵活性。除了对分布式数据执行复杂查询之外，还有第二个选择——在主要转换和过滤之后，使用 Sqoop 将数据从 Hadoop 采集出来，然后存入关系数据库中，以便能更直接地应用正常报告或 Tableau 可视化。因此，了解数据如何从许多较小的系统，流入较大的数据湖泊系统，再流回到较小的系统，是数据仓储的关键所在。

2. NoSQL方法：HBase

这里讨论的数据仓储的非关系选项是 HBase，一个列式的 NoSQL 数据库。列式数据库是 OLAP（online analytical processing，联机分析处理）类数据库访问的主力军。这类访问通常扫描大多数或所有数据库表，但只选择可用列的一部分。思考这样一个问题：每个地区每周有多少订单？这个订单表查询需要两列，分别是地区和订单日期。列式数据库只将这两列以紧凑、压缩的格式流入计算，而不采取每张表都逐行扫描（包括不需要的连接和列）的面向行的方法。因此，列式（也被称为以顶点为中心）计算为这些类型的聚合带来了巨大的性能提升。

在考虑使用哪种非关系工具和 NoSQL 数据库时，通常会有特定的要求帮你作出选择。例如，如果查询需要快速查找一个值，则应考虑键 / 值存储；如果数据访问需求涉及稀疏数据的行级写入，并且分析主要关注聚合，那么 HBase 是很好的备选工具；如果数据是实体（顶点）之间有许多关系（边）的图，则应考虑 Titan 这样的图数据库；如果你正在使用传感器或时间序列数据，那么应该考虑 InfluxDB 这样的原生了解时间序列数据的数据库。NoSQL 数据库的数量令人惊讶，这是因为它们通常都只针对特定的使用场景进行优化。在大多数情况下，这些数据存储后端是更大、更复杂的分布式存储和计算架构的一部分。

10.2　机器学习生命周期

第 5 章探索了分解数据集的抽样技术，将样本放在单个计算机上，然后使用 Scikit-Learn 来生成模型。可以序列化模型并通过分布式方法使用整个数据集对模型进行交叉验证。一般来说，这是一种非常有效的技术，被称为“最后一英里计算”。它使用 MapReduce 或 Spark 过滤、聚合或汇总数据，使数据可以加载到单个计算机的内存（例如 64GB）中，并能通过更容易获得的技术计算。此外，这也是执行没有分布式实现的计算或分析的唯一方法。

第 9 章探讨了如何使用 SparkML 库在分布式环境中执行分类、回归和聚类。过去，大数据机器学习依赖于 Mahout 库和图分析库（例如 Pregel）；而现在，SparkML 和 GraphX 库被广泛应用于分析上下文中。在一定程度上，将强大工具转换为分布式形式的趋势已经出现；但在其他情况下，分布式算法的出现比单进程版本还要早。

注 1：《Hadoop 应用架构》，Mark Grover 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1710。——编者注

鉴于之前已经定义了数据产品，所以希望大家明确一点：本书中讨论的所有数据管理技术都是趋向机器学习的，以特征工程的形式为主。特征工程是分析创建**决策空间**的过程；也就是为了创建一个有效的模型，需要什么维度（列或字段）？ 其实这个过程是数据科学家的主要工作；数据产品的最终目标是如何使用前面章节讨论的工具，而不是如何设计或开发它们。

因此，了解清楚机器学习算法在期望什么，可能比直接讨论机器学习更有用。

> 本书主要帮助数据科学家培养在大型数据集上进行机器学习特征工程的能力。几乎所有机器学习算法都在单个实例表上运行，每一行都是学习的实例，每一列都是决策空间中的一个维度。这对在数据产品生命周期中如何选择工具有很大影响。

在关系上下文中，这意味着数据集必须在分析之前**去规一化**（例如将多个表连接成一个表）。这可能会给系统带来冗余数据，但这正是算法所需要的。几乎所有机器学习系统都是迭代的，这意味着系统会多次处理数据。在大数据环境中，这将导致大量开销，因此我们会使用 Spark 替代 MapReduce 来进行机器学习——Spark 将数据保存在内存中，加快每次处理的速度。

去规一化、冗余和迭代算法也对数据生命周期有影响。如果我们经常生成单个表，那么就必须问问自己，为什么最初要从数据湖泊中规一化数据。难道就不能简单地将非规一化数据直接发送到机器学习模型中吗？在实践中，Hadoop 中的模式设计高度依赖于具体的分析过程或机器学习模型的输入需求。在许多情况下，可能有多个差异很小的数据模式需求，例如所需的分区或分桶模式。虽然使用不同的物理结构存储相同的数据集通常在传统数据仓库中被认为是一种反模式，但这种方法在 Hadoop 中是有意义的——数据针对一次写入优化，存储重复数据的开销也很小。[2]

考虑过机器学习构建阶段的数据存储后，第二件要思考的事是如何将模型从数据产品生命周期转移到生产中，以便将其用于识别模式、作出预测或适应用户行为！模型拟合数据，从而广泛应用于新的输入数据。拟合过程通常会产生一些模型的表示，可用于预测。例如，如果你使用朴素贝叶斯模型系列，那么拟合模型实际上是一组概率。通过这些概率中包含的实例特征的概率，我们可以计算类别的条件概率。如果你使用线性模型，那么拟合模型表示的是一组系数和一个截距，截距与自变量（特征）的线性组合产生一个因变量（目标）。

无论如何，这种表示必须从系统中导出才能运行和评估。线性模型的表示非常小，只是一组系数而已。贝叶斯模型的拟合模型可能更大一点——它是系统中每个特征和类的一组概率，因此模型表示的大小与特性数量直接相关。随机森林是多个决策树的集合，使用基于规则的方法分割决策空间。虽然每个决策树只是一个较小的树状数据结构，但是在大数据环境中，决策空间可能又大又复杂，随机森林中决策树的数量也可能带来存储问题。模型表示越来越大，一直到 k 最近邻方法——它需要存储每个用于距离计算的训练实例来作出决策。

注 2：《Hadoop 应用架构》，Mark Grover 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1710。——编者注

到目前为止，我们已经知道了导出拟合模型的两个主要机制：使用 Python 和 Scikit-Learn 序列化数据和将 Spark 模型写回 HDFS。但如果模型表示管理过程是数据产品生命周期的一部分，你会发现其他分析任务（规范化、删除重复数据、抽样等）也很有必要。

10.3 小结

大数据科学就是使用分布式计算技术进行描述性分析和推断性分析，希望这些需要分布式计算的数据的数量、多样性和速度将带来更深入或更有针对性的见解。此外，数据科学的产出是**数据产品**——从数据中获取价值并生成新数据的产品。因此，各种生态系统工具的集成通常围绕数据产品生命周期进行架构。

数据产品生命周期中有一个内部机器学习生命周期，它又包含两个主要阶段：**构建**阶段和**运行**阶段。构建阶段需要特征分析和数据探索；运行阶段旨在将产品的数据生成方面暴露给与数据产品交互的真实用户，生成可用于调整模型的数据，使模型更准确或更通用。通过提供数据采集、数据整理、数据探索和计算框架，数据产品生命周期提供了构建和运行模型的工作流。大多数生产架构结合了人工（由数据科学家驱动计算）分析和自动数据处理工作流，这些工作流由 Hadoop 技术生态系统提供和管理。

Hadoop 和分布式计算技术的生态系统是巨大且不断扩展的，但本书只讨论了它的基本概念，以及在评估和选择基于 Hadoop 的工具和算法实现数据产品工作流时要考虑的一些因素和作出的取舍——因为这都取决于你的需求。接下来该怎么做，是在自己的集群上进行实验和应用这些工具和模式，还是更深入地研究 Hadoop 或相关项目，都取决于你。但是希望本书介绍的概念、工具和技术为你提供了一个清晰的起点，并能持续为你的分布式数据分析之旅提供助力。

如果你读到这里了，那真的要恭喜你！你终于到达了 Hadoop 分布式数据分析指南的终点，希望这份指南是广泛且实用的。本书的目标是让你拥有足够的基础知识和背景知识，了解如何使用 Hadoop 分布式计算进行强大的大规模数据分析，并为深入了解其他一些子主题和技术作好准备。

附录A

创建Hadoop伪分布式开发环境

为了执行本书中的代码，你需要设置开发环境。Hadoop 开发人员通常在**伪分布式环境**（也被称为**单个节点设置**）上测试其脚本和代码，该虚拟机在单个机器上同时运行所有 Hadoop 守护进程。

如下指导将帮助你在 Ubuntu 14.04 上使用 Hadoop 2.5.0 安装一个伪分布式环境。

A.1 快速上手

如果你不熟悉 Linux 系统管理，或者不想自己去安装 Hadoop，那么有几个选择。我们提供了一个 VMDK，供你在选中的虚拟化软件（如 VirtualBox 或 VMWare Fusion）中使用；此外，Hortonworks 和 Cloudera 都提供了可快速下载的虚拟机。

若想快速安装，只需下载虚拟机并在你最喜欢的虚拟化软件中运行它。请注意，如果你使用 Cloudera 或 Hortonworks 的发行版，那么可能与我们使用的环境略有不同。要完成所有设置，请下载预先配置好的机器或按照如下所述步骤进行。

如果你使用了我们提供的 VMDK，请使用以下用户名和密码登录机器：

```
username: student
password: password
```

如果你有信心自己设置环境，那么请继续看下一节！

A.2 设置Linux环境

在开始安装 Hadoop 之前，你需要配置一个可以使用的 Linux 环境。下面的指导假设你能在你选的机器上安装 Ubuntu 14.04 发行版——要么选择双引导配置，要么选择虚拟机。你

可以凭喜好选择使用 Ubuntu 服务器版还是 Ubuntu 桌面版，但是不管怎样都需要熟悉如何使用命令行。

我们的基础环境是 Ubuntu x64 Desktop 14.04 LTS。

通过运行以下命令，确保你的系统是最新的：

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install build-essential ssh lzop git rsync curl
$ sudo apt-get install python-dev python-setuptools
$ sudo apt-get install libcurl4-openssl-dev
$ sudo easy_install pip
$ sudo pip install virtualenv virtualenvwrapper python-dateutil
```

A.2.1 创建Hadoop用户

为了保障 Hadoop 服务的安全，确保 Hadoop 作为 Hadoop 特定的用户和组运行。此用户将能够启动与集群中其他节点的 SSH 连接，但没有能损害运行着服务的操作系统的管理访问权限。实施 Linux 权限也有助于保障 HDFS 的安全，并且是准备安全计算集群的第一步。

本教程不适用于运行阶段；但对数据科学家而言，这些权限终究能帮你减少一些麻烦，因此在开发环境中设置权限还是有用的。而且，这还能确保 Hadoop 与其他软件应用程序是分开的，有助于机器维护的条理性。

创建 `hadoop` 用户和组，然后将 `student` 用户添加到 Hadoop 组中：

```
$ sudo addgroup hadoop
$ sudo adduser --ingroup hadoop hadoop
$ sudo usermod -a -G hadoop student
```

注销并重新登录（或重新启动计算机），输入 `groups` 命令后，就能看到你已经加入了 Hadoop 组。

A.2.2 配置SSH

SSH 是必需的，必须安装在系统上才能使用 Hadoop（并且能更好地管理虚拟环境，如果你使用的是无界面的 Ubuntu 就更是如此）。通过以下命令为 `hadoop` 用户生成 `ssh` 密钥：

```
$ sudo su hadoop
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa):
Created directory '/home/student/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/student/.ssh/id_rsa.
Your public key has been saved in /home/student/.ssh/id_rsa.pub.
[... snip ...]
```

对所有提示输入都按回车键才能接受默认值，并创建不需要密码进行身份验证的密钥（这是 Hadoop 必需的）。由于需要没有密码的 SSH，将 Hadoop 用户与管理用户分开

是一种很好的做法；但因为这是一个开发集群，我们将选取捷径，将 student 用户作为 Hadoop 用户。

为了让 SSH 能使用密钥，使用以下命令将公钥复制到 authorized_keys 文件中：

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys
```

你应该可以下载这个密钥，并将其用于 SSH 连接 Ubuntu 环境。要测试 SSH 密钥，可使用以下命令：

```
$ ssh -l hadoop localhost
```

如果没有要求你输入密码就完成了，那么 Hadoop 的 SSH 就已经配置成功。

A.2.3　安装Java

Hadoop 和大多数 Hadoop 生态系统需要 Java 来运行。Hadoop 需要 Oracle Java 1.6.*x* 或更高版本，通常推荐使用特定版本的 Java。不过，Hadoop 现在维护了一份报告，列出了与 Hadoop 配合良好的各种 JDK。Ubuntu 没有在 Ubuntu 仓库中维护 Oracle JDK，因为它是专利代码，所以我们将安装 OpenJDK。有关支持的 Java 版本的更多信息，请参见 Hadoop Java 版本（http://wiki.apache.org/hadoop/HadoopJavaVersions）；有关在 Ubuntu 上安装不同版本的信息，请参见在 Ubuntu 上安装 Java（https://help.ubuntu.com/community/Java）。

```
$ sudo apt-get install openjdk-7-*
```

进行快速检查，确保安装了正确版本的 Java：

```
$ java -version
java version "1.7.0_55"
OpenJDK Runtime Environment (IcedTea 2.4.7) (7u55-2.4.7-1ubuntu1)
OpenJDK 64-Bit Server VM (build 24.51-b03, mixed mode)
```

目前，Hadoop 已在 OpenJDK 和 Oracle 的 JDK/JRE 上进行了构建和测试。

A.2.4　禁用IPv6

有报告称 Ubuntu 上运行的 Hadoop 与 IPv6 有冲突。因此，自 Hadoop 0.20 以后，Ubuntu 用户一直在其集群中禁用 IPv6。虽然目前尚不清楚最新版本的 Hadoop 是否仍然有这个漏洞，但是在单个节点或者说伪分布式环境中无须使用 IPv6，因此最好禁用它，避免任何潜在问题。

通过执行以下代码行，编辑 /etc/sysctl.conf 文件：

```
$ gksu gedit /etc/sysctl.conf
```

然后将以下内容添加到文件的末尾：

```
# 禁用ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

要使更改生效，请重新启动计算机。一旦重新启动，请使用以下命令检查状态：

```
$ cat /proc/sys/net/ipv6/conf/all/disable_ipv6
```

如果输出为 0，则 IPv6 是启用的；如果为 1，说明已成功禁用 IPv6。

A.3 安装Hadoop

要获取 Hadoop，你需要从其中一个 Apache 下载镜像（http://www.apache.org/dyn/closer.cgi/hadoop/common/）下载你选择的版本。如下指导将下载编写本书时带 YARN 的 Hadoop 稳定版本——Hadoop 2.5.0。

选择镜像后，在终端窗口中键入以下命令，将 http://apache.mirror.com/hadoop-2.5.0/ 替换为你选择的、最适合你所在区域的镜像 URL：

```
$ curl -O http://apache.mirror.com/hadoop-2.5.0/hadoop-2.5.0.tar.gz
```

通过确保 `md5sum` 与镜像中的 `md5sum` 匹配来验证下载：

```
$ md5sum hadoop-2.5.0.tar.gz
5d5f0c8969075f8c0a15dc616ad36b8a hadoop-2.5.0.tar.gz
```

当然，你也可以使用任何你想用的办法下载 Hadoop——使用 `wget` 命令或浏览器都可以。

A.3.1 解压

在得到压缩的 tarball 文件之后，下一步就是将它解压。你可以使用 Archive Manager，或者按照以下说明进行操作。你必须作出一个最重要的决定：要将 Hadoop 解压到何处。

Linux 操作系统依赖于分层目录结构。在根目录下，你听说过的许多目录都有特定用途：/etc 用于存储配置文件，而 /home 用于存储用户特定的文件。大多数应用程序遍布在多个位置。例如，/bin 和 /sbin 中有对操作系统至关重要的程序；/usr/bin 和 /usr/sbin 中的程序虽不是至关重要，但是系统范围的。/usr/local 用于本地安装的程序，而 /var 用于包含缓存和日志的程序数据。你可以在这篇 Stack Exchange 文章（http://bit.ly/1Tr6QuW）中了解更多有关这些目录的信息。

将 Hadoop 移至 /opt 和 /srv 目录是不错的选择。/opt 包含非打包程序，通常为源码，很多开发人员将他们的代码放在那里用于部署。/srv 目录代表服务；Hadoop、HBase、Hive 等作为服务在机器上运行，所以这似乎也是一个好地方。此外，它还是一个很容易访问的标准位置，因此我们把所有内容都放在这里。输入以下命令：

```
$ tar -xzf hadoop-2.5.0.tar.gz
$ sudo mv hadoop-2.5.0 /srv/
$ sudo chown -R hadoop:hadoop /srv/hadoop-2.5.0
$ sudo chmod g+w -R /srv/hadoop-2.5.0
$ sudo ln -s /srv/hadoop-2.5.0 /srv/hadoop
```

这些命令解压 Hadoop，将其移至服务目录，然后设置权限。所有 Hadoop 和集群服务将存放在服务目录。最后，创建一个我们希望使用的 Hadoop 版本的 symlink，以便将来可以轻松升级 Hadoop 发行版。

A.3.2 环境

为了确保一切正常执行，来设置一些环境变量，确保 Hadoop 在正确的上下文中执行。在命令行中输入以下命令，使用 hadoop 用户配置文件打开文本编辑器来更改环境变量：

```
$ gksu gedit /home/hadoop/.bashrc
```

将如下内容添加到该文件：

```
# 设置Hadoop相关的环境变量
export HADOOP_HOME=/srv/hadoop
export PATH=$PATH:$HADOOP_HOME/bin

# 设置JAVA_HOME
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```

我们还将添加一些方便的功能到 student 用户环境。使用以下命令打开 student 用户 bash 别名文件：

```
$ gedit ~/.bash_aliases
```

将以下内容添加到该文件：

```
# 设置Hadoop相关的环境变量
export HADOOP_HOME=/srv/hadoop
export HADOOP_STREAMING=$HADOOP_HOME/share/hadoop/tools/lib/
hadoop-streaming-2.5.0.jar
export PATH=$PATH:$HADOOP_HOME/bin

# 设置JAVA_HOME
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64

# 有用的别名
alias ..="cd .."
alias ...="cd ../.."
alias hfs="hadoop fs"
alias hls="hfs -ls"
```

这些简单的别名可以节省大量的打字时间！ 随意添加任何你认为在开发工作中可能有用的配置。

通过运行 Hadoop 命令，检查你的环境配置是否有效：

```
$ hadoop version
Hadoop 2.5.0
Subversion http://svn.apache.org/repos/asf/hadoop/common -r 1616291
Compiled by jenkins on 2014-08-06T17:31Z
Compiled with protoc 2.5.0
From source with checksum 423dcd5a752eddd8e45ead6fd5ff9a24
This command was run using /srv/hadoop-2.5.0/share/hadoop/common/
hadoop-common-2.5.0.jar
```

如果没有出现错误，并显示类似于此处的输出，则所有内容都已正确配置。

A.3.3 Hadoop配置

设置伪分布式 Hadoop 的倒数第二步是编辑 Hadoop 环境、MapReduce site、HDFS site 和 YARN site 的配置文件。这主要指的是编辑配置文件。

通过在命令行中输入以下内容来编辑 hadoop-env.sh 文件：

```
$ gedit $HADOOP_HOME/etc/hadoop/hadoop-env.sh
```

该配置最重要的部分是更改以下内容：

```
# 使用的Java实现
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```

接着，编辑 core site 配置文件：

```
$ gedit $HADOOP_HOME/etc/hadoop/core-site.xml
```

使用如下内容替换 `<configuration></configuration>`：

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/var/app/hadoop/data</value>
    </property>
</configuration>
```

下一步，编辑 `mapreduce site` 配置——通过复制模板，打开文件进行编辑：

```
$ cp $HADOOP_HOME/etc/hadoop/mapred-site.xml.template \
    $HADOOP_HOME/etc/hadoop/mapred-site.xml
$ gedit $HADOOP_HOME/etc/hadoop/mapred-site.xml
```

使用如下内容替换 `<configuration></configuration>`：

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
     </property>
</configuration>
```

现在通过编辑如下文件，来编辑 `hdfs site` 配置：

```
$ gedit $HADOOP_HOME/etc/hadoop/hdfs-site.xml
```

使用如下内容替换 `<configuration></configuration>`：

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
```

```
    </property>
</configuration>
```

最后，编辑 yarn site 文件：

```
$ gedit $HADOOP_HOME/etc/hadoop/yarn-site.xml
```

按如下所示更新配置文件：

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>localhost:8025</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>localhost:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>localhost:8050</value>
    </property>
</configuration>
```

编辑完这些文件后，Hadoop 就已被全副武装，成为了一个伪分布式环境。

A.3.4 格式化NameNode

启动 Hadoop 前的最后一步是格式化 NameNode。NameNode 负责管理 HDFS。这台机器上的 NameNode 将把它的文件保存在 /var/app/hadoop/data 目录中。我们需要初始化这个目录，然后格式化 NameNode 来正确使用它：

```
$ sudo mkdir -p /var/app/hadoop/data
$ sudo chown hadoop:hadoop -R /var/app/hadoop
$ sudo su hadoop
$ hadoop namenode -format
```

你会看到页面向下滚动显示了一大堆 Java 消息。如果 namenode 命令成功执行（/var/app/hadoop/data 目录下应该有目录，包括一个 dfs 目录），那么 Hadoop 就设置完成并可以使用了！

A.3.5 启动Hadoop

此时就可以启动和运行 Hadoop 守护进程了。格式化 NameNode 时，用 sudo su hadoop 命令切换到 hadoop 用户。如果还是该用户，继续执行以下命令：

```
$ $HADOOP_HOME/sbin/start-dfs.sh
$ $HADOOP_HOME/sbin/start-yarn.sh
```

守护进程会启动并发出消息，内容包括记录日志的位置和其他重要信息。如果遇到有关 SSH 密钥的问题，只需要在遇到提示时输入 `y`。可以通过 `jps` 命令查看正在运行的进程：

```
$ jps
5298 Jps
4690 ResourceManager
4541 SecondaryNameNode
4813 NodeManager
4227 NameNode
```

如果进程没有运行，就一定有哪里出错了。你可以打开浏览器，访问 http://localhost:8088 来访问 Hadoop 集群管理网站——这会打开一个页面，上面有 Hadoop 标志和一个应用列表。

最后，在 HDFS 上为 student 账户准备一个空间，用来存储数据并运行分析作业：

```
$ hadoop fs -mkdir -p /user/student
$ hadoop fs -chown student:student /user/student
```

现在可以使用 `exit` 命令退出 `hadoop` 用户的 shell。

A.3.6 重启Hadoop

如果重新启动机器，Hadoop 守护进程将停止运行，并且不会自动重新启动。如果你尝试运行 Hadoop 命令，并且收到了消息“Connection refused”，则可能是由于守护进程未运行。可以使用 `sudo` 运行 `jps` 命令来进行检查：

```
$ sudo jps
```

要在关闭后重新启动 Hadoop，运行以下命令即可：

```
$ sudo -H -u hadoop $HADOOP_HOME/sbin/start-dfs.sh
$ sudo -H -u hadoop $HADOOP_HOME/sbin/start-yarn.sh
```

这些进程就会作为 `hadoop` 用户重新启动，你又可以继续了！

附录B

安装Hadoop生态系统产品

除了Hadoop提供的核心功能之外，本书也涵盖了构建于Hadoop之上的其他Hadoop生态系统项目。在典型设置中，这些产品通常要么安装在运行Hadoop和YARN的集群上，要么通过配置连接到Hadoop集群。本书假设你已经在单个节点上设置和配置了伪分布式模式的Apache Hadoop。然而，要运行单节点Hadoop集群和Hadoop生态系统产品，还有其他几个选择，本书也将对此进行讨论。

B.1 打包的Hadoop发行版

要运行Hadoop单机配置，最简单的方法是安装一款由主流Hadoop厂商提供的虚拟化Hadoop发行版，比如Cloudera的Quickstart VM（ttp://bit.ly/1YWtzPC）、Hortonworks Sandbox（http://bit.ly/1YWtyLy）和MapR的Hadoop沙箱（http://bit.ly/1YWtz27）。除了一个单节点Hadoop集群之外，这些虚拟机还包含流行的Apache Hadoop生态系统项目以及专有的应用程序和工具，它们都在一个简单的完整包（turn-key bundle）里。你可以使用喜欢的虚拟化软件，如VMWare Player（https://www.vmware.com/products/player）或Virtualbox（https://www.virtualbox.org/wiki/Downloads）来运行这些虚拟机。

B.2 自己安装Apache Hadoop生态系统产品

如果你没有使用打包的Hadoop发行版，而是手动安装了Apache Hadoop，那么你就还需要手动安装和配置本书中讨论的各种Hadoop生态系统项目，使它们能使用安装好的Hadoop。

在大多数情况下，在设置好的Hadoop环境中安装服务（例如Hive、HBase等）包含如下几个步骤。

(1) 下载该服务的 tarball 发布文件。
(2) 将发布文件解压到 /srv/ 目录（其中安装了 Hadoop 服务），并从发布文件创建一个符号链接指向一个简单的名称。
(3) 配置环境变量，添加服务的路径。
(4) 配置服务，以伪分布式模式运行。

在本附录中，我们将逐步安装 Sqoop，从而使用伪分布式 Hadoop 集群。这些步骤同样适用于几乎所有本书讨论的 Hadoop 生态系统项目。

B.2.1 基本安装和配置步骤

首先从 Apache Sqoop 下载镜像（http://www.apache.org/dyn/closer.cgi/sqoop/1.4.6）下载最新的 Sqoop 稳定版本，本书创作时的版本为 1.4.6。确保你是拥有管理员（`sudo`）权限的用户，并获取与 Hadoop 版本（在本示例中为 Hadoop 2.5.1）兼容的 Sqoop 版本：

```
~$ wget http://apache.arvixe.com/sqoop/1.4.6/sqoop-1.4.6.bin__
hadoop-2.0.4-alpha.tar.gz
~$ sudo mv sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz /srv/
~$ cd /srv
/srv$ sudo tar -xvf sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz
/srv$ sudo chown -R hadoop:hadoop sqoop-1.4.6.bin__hadoop-2.0.4-alpha
/srv$ sudo ln -s $(pwd)/sqoop-1.4.6.bin__hadoop-2.0.4-alpha $(pwd)/sqoop
```

现在使用 `sudo su` 命令切换到 `hadoop` 用户，编辑你的 Bash 配置，添加一些能提供方便的环境变量：

```
/srv$ sudo su hadoop
$ vim ~/.bashrc
```

将以下环境变量添加到你的 bashrc 配置文件中：

```
# Sqoop别名
export SQOOP_HOME=/srv/sqoop
export PATH=$PATH:$SQOOP_HOME/bin
```

然后使用 `source` 运行配置文件，将新变量添加到当前 shell 环境中：

```
~$ $ source ~/.bashrc
```

通过从 `$SQOOP_HOME` 运行 `sqoop help` 来验证 Sqoop 是否安装成功：

```
/srv$ cd $SQOOP_HOME
/srv/sqoop$ sqoop help

15/06/04 21:57:40 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen            Generate code to interact with database records
  create-hive-table  Import a table definition into Hive
  eval               Evaluate a SQL statement and display the results
  export             Export an HDFS directory to a database table
  help               List available commands
```

```
  import               Import a table from a database to HDFS
  import-all-tables    Import tables from a database to HDFS
  job                  Work with saved jobs
  list-databases       List available databases on a server
  list-tables          List available tables in a database
  merge                Merge results of incremental imports
  metastore            Run a standalone Sqoop metastore
  version              Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

如果你看到任何与 HCatalog 有关的警告，暂且可以放心地忽略它们。由代码可知，Sqoop 提供了一系列导入、导出特定的命令和工具，能与数据库或 Hadoop 数据源连接。

Sqoop 进程要么通过运行 Sqoop 命令手动执行，要么由上游系统调度或触发 Sqoop 操作来执行。但是，我们将要安装的其他产品还包括启动守护进程的命令。可以使用 `jps` 命令将这些运行中的进程列出，比如所有的 Java 进程。`jps` 命令在验证所有预期的 Hadoop 进程是否运行时非常有用；例如，如果你按照附录 A 所述启动了 Hadoop，应该能看到以下进程：

```
~$ jps
10029 NameNode
10670 NodeManager
21694 Jps
10187 DataNode
10373 SecondaryNameNode
11034 JobHistoryServer
10541 ResourceManager
```

如果没有看到这些进程，请参见附录 A 和第 2 章中有关启动和停止 Hadoop 服务的讨论。

B.2.2　Sqoop特定配置

在将 MySQL 表数据导入 HDFS 之前，需要下载 MySQL JDBC 连接器驱动程序，并将其添加到 Sqoop 的 lib 文件夹中：

```
~$ wget http://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-5.1.
30.tar.gz
~$ tar -xvf mysql-connector-java-5.1.30.tar.gz
~$ cd mysql-connector-java-5.1.30
$ sudo cp mysql-connector-java-5.1.30-bin.jar /srv/sqoop/lib/
$ cd $SQOOP_HOME
```

这让 Sqoop 能连接到我们的 MySQL 数据库。你现在应该已经在本地开发环境中成功安装了 Sqoop 和 MySQL 服务器和客户端，并配置好了 Sqoop，可以导入和导出 MySQL。

B.2.3　Hive特定配置

Hive 的安装类似于 Sqoop。但一旦安装了 Hive，就需要将其配置为在 Hadoop 单节点集群上运行。具体来说，Hive 要求配置 Hive 仓库（将包含 Hive 的数据文件）和 metastore 数据库（将包含 Hive 的模式和表的元数据）。

1. Hive仓库目录

默认情况下，Hive 数据存储在 HDFS 中，位于 /user/hive/warehouse 仓库目录。我们需要确保 HDFS 中有这个目录，而且所有 Hive 用户都可以写入。如果要更改此位置，可以通过覆盖 $HIVE_HOME/conf/hive-site.xml 中的配置来修改 `hive.metastore.warehouse.dir` 属性的值。

至于单节点配置，假设我们将使用默认仓库目录，并在 HDFS 中创建必要的目录。首先创建一个 /tmp 目录、一个 hive 用户目录和默认仓库目录：

```
$ hadoop fs -mkdir /tmp
$ hadoop fs -mkdir -p /user/hive
$ hadoop fs -mkdir /user/hive/warehouse
```

还需要设置这些目录的权限，以便它们可以由 Hive 写入：

```
$ hadoop fs –chmod g+w /tmp
$ hadoop fs –chmod g+w /user/hive/warehouse
```

此外，Hive 得能写入你配置的本地 Hadoop 临时数据目录。因此，还需要确保 `hadoop` 组有写入权限，以在该路径中创建目录：

```
$ chmod g+w /var/app/hadoop/data
```

2. Hive metastore数据库

Hive 需要一个 metastore 服务后端，供它用于存储表模式定义、分区和相关元数据。Hive metastore 服务还通过 metastore 服务 API 为客户端（包括 Hive）提供访问 metastore 信息的接口。

配置 metastore 有几种不同的方式，默认的 Hive 配置使用嵌入的 metastore Derby SQL Server，提供单进程存储，Hive 驱动程序、metastore 接口和 Derby 数据库共享相同的 JVM。这个配置便于开发和单元测试，但不支持真正的集群配置，因为任何时候都只有单个用户可以连接到 Derby 数据库。能用于生产的配置将包含 MySQL 或 PostgreSQL 等数据库。

出于本章的目的，我们将使用嵌入的 Derby 服务器作为 metastore 服务。但是建议你阅读 Apache Hive 手册，安装生产级配置的本地或远程 metastore 服务器。

默认情况下，Derby 将在启动 Hive 会话的当前工作目录下创建一个 metastore_db 子目录。但如果你更改了工作目录，Derby 将无法找到以前的 metastore，并会重新创建它。为了避免这种情况发生，需要更新 metastore 配置，对 metastore 数据库的永久位置进行配置：

```
~$ cd $HIVE_HOME/conf
/srv/hive/conf$ sudo cp hive-default.xml.template hive-site.xml
/srv/hive/conf$ vim hive-site.xml
```

查找名称为 `javax.jdo.option.ConnectionURL` 的属性，将其更新为绝对路径：

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby:;databaseName=/home/hadoop/metastore_db;create=true</value>
    <description>JDBC connect string for a JDBC metastore</description>
</property>
```

更新 `ConnectionURL` databaseName 后，保存并关闭该文件。

3. 验证Hive正在运行

现在可以从 Hive 的安装目录启动预打包的 Hive 命令行接口（CLI），验证 Hive 配置是否正确，以及它能否在伪分布式 Hadoop 集群上运行了。

从 $HIVE_HOME 目录启动 Hive CLI：

```
~$ cd $HIVE_HOME
/srv/hive$ bin/hive
```

如果 Hive 配置正确，该命令将启动 CLI 并显示 Hive CLI 提示：

```
hive>
```

你可能会看到与 Hive metastore 配置过时有关的警告：

```
WARN conf.HiveConf: DEPRECATED: hive.metastore.ds.retry.* no longer has any
effect.
Use hive.hmshandler.retry.* instead
```

但是，如果你看到任何错误，请根据之前的建议检查配置，然后重试。可以使用以下命令随时退出 Hive CLI：

```
hive> exit;
```

现在可以在本地和伪分布式模式下使用 Hive 来运行 Hive 脚本了。

B.2.4　HBase特定配置

HBase 在安装后需要一些额外的配置。与 Sqoop 和 Hive 不同，它需要启动守护进程才可以与我们进行交互。

解压并安装 HBase 之后，它的目录中有一个包含 HBase 配置文件的 /conf 目录。我们将编辑配置文件 conf/hbase-site.xml，将 HBase 配置为使用 HDFS 以伪分布式模式运行，并将 ZooKeeper 文件写入本地目录。使用 `vim` 编辑 HBase 配置文件：

```
$ vim $HBASE_HOME/conf/hbase-site.xml
```

然后覆盖配置文件的以下三个配置：

```
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>hdfs://localhost:9000/hbase</value>
    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/home/hadoop/zookeeper</value>
    </property>
</configuration>
```

通过此配置，HBase 将启动一个 HBase Master 进程、一个 ZooKeeper 服务器和一个 RegionServer 进程。默认情况下，HBase 将所有目录配置为 /tmp，这意味着除非更改配置，否则服务器重新启动时将丢失所有数据，因为大多数操作系统重启时都会清除 /tmp。通过更新 `hbase.zookeeper.property.dataDir` 属性，HBase 会将数据写入 hadoop 主目录下的可靠数据路径中。

> HBase 需要有对本地目录的写入权限才能维护 ZooKeeper 文件。因为我们将作为 `hadoop` 用户运行 HBase（或者你设置的任何启动 HDFS 和 YARN 的用户），所以请确保 `dataDir` 被配置为了 Hadoop 用户可以写入的路径（例如 /home/hadoop）。

还需要更新 HBase 环境设置的 JAVA_HOME 路径。为此，在 conf/hbase-env.sh 中取消注释并修改以下设置：

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle
```

现在，HBase 应该配置正确，能在单节点集群上以伪分布式模式运行了。

启动HBase

现在就可以启动 HBase 进程了。但在此之前，应该确保 Hadoop 正在运行：

```
/srv/hbase$ jps
4051 NodeManager
3523 DataNode
3709 SecondaryNameNode
3375 NameNode
9436 Jps
3921 ResourceManager
```

如果 HDFS 和 YARN 进程没在运行，先使用 $HADOOP_HOME/sbin 下的脚本启动它们。

现在可以启动 HBase 了！

```
/srv/hbase$ bin/start-hbase.sh
localhost: starting zookeeper, logging to /srv/hbase/bin/../logs/
hbase-hadoop-zookeeper-ubuntu.out
starting master, logging to /srv/hbase/logs/
hbase-hadoop-master-ubuntu.out
localhost: starting regionserver, logging to
/srv/hbase/bin/../logs/hbase-hadoop-regionserver-ubuntu.out
```

可以使用 `jps` 命令来验证哪些进程正在运行，该命令将展示正在运行的 Hadoop 进程、HBase 和 ZooKeeper 进程、HMaster、HQuorumPeer 以及 HRegionServer：

```
/srv/hbase$
4051 NodeManager
10225 Jps
3523 DataNode
3709 SecondaryNameNode
3375 NameNode
3921 ResourceManager
```

```
9708 HQuorumPeer
9778 HMaster
9949 HRegionServer
```

可以随时使用 stop-hbase.sh 脚本停止 HBase 和 ZooKeeper：

```
/srv/hbase$ bin/stop-hbase.sh
stopping hbase.................
HBase Shell
```

HBase 启动之后，可以使用 HBase shell 连接到正在运行的实例：

```
/srv/hbase$ bin/start-hbase.sh
/srv/hbase$ bin/hbase shell
```

你会收到提示：

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.9-hadoop2, r96878ece501b0643e879254645d7f3a40eaf101f,
Mon Dec 15 23:00:20 PST 2014

hbase(main):001:0>
```

有关 HBase shell 支持的命令的文档，请使用 `help` 获取命令列表：

```
hbase(main):001:0> help
```

还可以使用 `status` 命令检查 HBase 集群的状态：

```
hbase(main):002:0> status
1 servers, 0 dead, 3.0000 average load
```

要退出 shell，只需使用 `exit` 命令：

```
hbase(main):003:0> exit
```

你现在可以在伪分布式模式下使用 HBase 了。一定要记住，在与 HBase shell 进行交互之前，必须先启动并运行 Hadoop 进程和 HBase 进程。

B.2.5 安装Spark

在本地机器上设置和运行 Spark 非常简单，一般遵循安装其他 Hadoop 生态系统的模式。按照伪分布式 Ubuntu 机器的说明，我们已经满足了 Spark 的主要要求，即 Java 7+ 和 Python 2.6+。确保 `java` 和 `python` 程序在路径上，并且设置了 `$JAVA_HOME` 环境变量（如前所述）。

在之前的安装说明中，我们使用 `wget` 或 `curl` 直接从 Apache 镜像获取了 tarball 文件。但对于 Spark 来说，情况略有不同。打开浏览器并按照以下步骤下载正确版本的 Spark。

(1) 进入 Spark 下载页面（http://spark.apache.org/downloads.html）。
(2) 选择最新的 Spark 版本（在创作本书时为 1.5.2），并确保选择了一个针对 Hadoop 2.4 或更高版本预构建的软件包，直接下载。

Spark 的版本更新往往比较频繁。为了确保可以下载到新版本的 Spark 并立即使用它们，我们将把 Spark 软件包解压到服务目录中，然后将该版本符号链接到一个泛化的 spark 目录。要更新版本的话，只需下载最新版本，并将符号链接重定向到它即可。通过这种方式，新版本也将拥有所有的环境变量和配置！

首先，遵循标准惯例来安装 Hadoop 生态系统服务：

```
$ tar -xzf spark-1.5.2-bin-hadoop2.4.tgz
$ mv spark-1.5.2-bin-hadoop2.4 /srv/spark-1.5.2
```

然后，创建 Spark 的符号链接版本：

```
$ ln -s /srv/spark-1.5.2 /srv/spark
```

编辑 Bash 配置文件，将 Spark 添加到 `$PATH` 并设置 `$SPARK_HOME` 环境变量。如前所述，我们将切换到 Hadoop 用户，但你也可以将以上所有内容添加到 `student` 用户的配置文件中：

```
$ sudo su hadoop
$ vim ~/.bashrc
```

将以下内容添加到配置文件：

```
export SPARK_HOME=/srv/spark
export PATH=$SPARK_HOME/bin:$PATH
```

接下来，使用 `source` 运行配置文件（或重新启动终端），将这些新变量添加到环境中。完成之后，你应该能够运行一个本地 `pyspark` 解释器：

```
$ pyspark
Python 2.7.10 (default, Jun 23 2015, 21:58:51)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
[… snip …]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.2
      /_/

Using Python version 2.7.10 (default, Jun 23 2015 21:58:51)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

至此，Spark 已成功安装，可在本地计算机上以独立模式运行，也足以运行本书的例子了。如果你想测试 Spark/Hadoop 连接，还可以使用 `spark-submit` 将作业直接提交给以伪分布式模式运行的 YARN 资源管理器。有关此主题和其他主题的更多信息，比如在 EC2 上使用 Spark 或使用 iPython notebook 设置 Spark，请参阅 Benjamin Bengfort 的"Getting Started with Spark (in Python)"（http://districtdatalabs.silvrback.com/getting-started-with-spark-in-python）。

最大限度简化Spark

Spark（和 PySpark）的执行可能非常冗长，会有许多 INFO 日志消息打印到屏幕上。这

在开发过程中特别烦人，因为 Python 堆跟踪或 `print` 语句的输出可能会丢失。为了降低 Spark 的冗长度，可以在 $SPARK_HOME/conf 中配置 log4j 设置，如下所示：

```
$ cp $SPARK_HOME/conf/log4j.properties.template \
      $SPARK_HOME/conf/log4j.properties
$ vim $SPARK_HOME/conf/log4j.properties
```

编辑 log4j.properties 文件，将代码中的每一行的 `INFO` 替换成 `WARN`，类似于：

```
# 设置:让所有事件记录到控制台
log4j.rootCategory=WARN, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}:
%m%n

# 减少输出第三方冗杂的日志
log4j.logger.org.eclipse.jetty=WARN
log4j.logger.org.eclipse.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=WARN
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=WARN
```

再次运行 PySpark，输出消息应该简洁多了！

术语表

可访问的

在一个计算集群上下文中，如果一个节点可以通过网络到达，那么它就是可访问的；在其他上下文中，如果一个工具或者库能轻易为特定人群使用和理解，那么它就是可访问的。

累加器

一个共享变量，只能应用满足结合律的运算，如加法（特定于 Spark，在 MapReduce 中称为计数器）。因为满足结合律的运算是与顺序无关的，所以无论运算顺序如何，累加器都可以在分布式环境中保持一致。

动作和转换

请参见“转换和动作”。

代理

代表用户例行运行的服务，通常是后台进程，独立执行任务。Flume 代理是构建数据流的基本单元，它从源中采集和整理数据，最终通过通道将数据传输到数据槽。

匿名函数

没有指定识别符（变量名称）的函数。这些函数通常在运行时构造，并作为参数传递给高阶函数；也可以用它们轻松创建闭包。传递匿名函数给 Spark 操作来定义它们的行为。另请参见“闭包”和“lambda 函数”。

应用程序编程接口（application programming interface，API）

用于指定软件组件如何交互的例程、协议和接口的集合。MapReduce API 指定用于构建 `Mapper`、`Reducer` 和 `Job` 子类的接口，定义 MapReduce 行为。与之类似，Spark 也有可以应用于 RDD 的转换和动作的 API。

ApplicationMaster

在 YARN 中，ApplicationMaster 是特定于框架的库（例如本书中的 MapReduce、Spark 或 Hive）的实例。ApplicationMaster 从 ResourceManager 协商取得资源，在 NodeManager 上执行进程，跟踪作业状态并监视进度。

满足结合律的

在数学中，不管满足结合律的运算如何分组，只要顺序保持不变，计算结果便相同。满足结合律的运算在分布式环境中很重要，因为它让你能在计算最终整体之前让多个处理器同时计算分组子操作。

Avro

Apache Avro 是 Apache Hadoop 内开发的一个远程过程调用（remote procedure call，RPC）数据序列化框架，它使用 JSON 定义模式和类型，然后以紧凑的二进制格式对数据进行序列化。

bag of words

在文本处理中，bag of words 是一个模型，根据最重要的令牌或单词的频率或出现编码文档，不考虑令牌的顺序 。

偏差

在机器学习中，因偏差引起的误差是模型的预期平均预测与正确值之间的差异。偏差通常用来衡量模型的错误程度。随着偏差的增大，方差逐渐减小。另请参见“方差”。

大数据

利用极大数据集探寻与人类行为和交互特别相关的模式、趋势和关系的计算方法学。大数据具体指的是海量、难处理和稍纵即逝的数据，单台机器无法进行可靠计算。 因此，大数据技术主要利用分布式计算和数据库技术来计算结果。

二元短语（bigram）

字符串或数组中的两个连续令牌的序列。令牌通常是字母、音节或单词。二元短语是 *n*-grams 的特定形式，其中 $n = 2$。

块

块是 HDFS 存储大文件的方法，将大文件分割成相同大小（通常为 128MB）的数据块。块在多个 DataNode 上都有副本（默认复制因子为 3），通过冗余提供数据持久性，并支持数据本地计算。

布隆过滤器

一个紧凑的概率数据结构，可用于测试某些数据是否是集合的成员。可能存在误算率（以为元素是集合的成员，而实际上不是），但可以通过调整过滤器的大小来设置误算出现的概率。漏报（以为元素不是集合的成员，而实际上是的）不会出现，因此布隆过滤器的召回率高达 100%。

广播变量

广播变量在 Spark 中是一种机制，用于创建按需传输到集群中每个节点的只读数据结构。广播变量可以包含计算所需的额外信息、先前转换的结果或查找表。因为它们是只

读的，所以是集群安全的。另请参见“分布式缓存”。

构建阶段

在机器学习中，构建阶段通常通过一些迭代优化过程将模型形式拟合到现有数据。构建阶段包括特征提取、特征变换，以及正则化或超参数调整。构建阶段的输出是拟合模型，可用于进行预测。

字节数组

由固定长度的单字节数组构成的数据结构，可以存储任何类型的信息（数字、字符串、文件的内容），并且非常一般化，因此作为行键用于 HBase。另请参见“行键”。

cascading

Driven, Inc. 的不关心规模的数据应用程序开发框架，为 MapReduce 提供了高级抽象。通常用于将数据流或多部分作业定义为有向无环图。

集中管理集群

一个拥有角色不同的两个节点的计算集群，分别是 manager（master 节点）和 worker 节点。集群中有一个或多个协调管理节点，集中决策，无须共识。管理节点负责数据的完整性、协调性、一致性，并处理客户端请求。另请参见“对等集群”。

质心

在无监督机器学习（聚类）中，质心是特征空间中的一个点，定义了集群的中心。尽管不是所有聚类算法都有质心（生成中心），但是那些有质心的算法将质心定义为簇中所有点的距离的平均。

通道

在计算机科学中，通道指信息流动的途径。这里指的是 Apache Flume 中的通道，它们是被动存储区或缓冲区，保存事件信息直到它们被下游数据槽收集。

点击率

衡量以引导用户浏览额外信息为目的的电子邮件、网页或广告的有效性。当用户单击超链接时，处理超链接请求的服务器会写入一条日志记录，从而获取点击率。例如，可以测量购物车应用程序中“购买”按钮的点击率。

client

一般来说指某个计算服务或资源的请求者，通常是人类用户。本书提到的 client 指发送 Web 请求、提交 MapReduce 作业的人，或 Spark 应用程序中驱动程序所在的计算机。也可以由代理代表 client 进行日常工作。

闭包

一个绑定到自己的封闭执行环境的函数，环境中有约束变量。因为环境将被赋予的变量映射到外部函数，所以这些变量不能被外部进程修改，这使得闭包可用于分布式上下文。

云计算

使用远程数据中心的共享计算资源（而不是利用本地服务器或个人设备）。共享计算资源通常是弹性的，这意味着你可以根据需要扩大和收缩资源的使用和分配。

集群

通常指执行集体或相关计算的设备的集合。在 Hadoop 中，指运行 HDFS 和 YARN 守护程序的一组服务器或计算机。

系数

在线性模型中，系数是定义因变量空间中的超平面的数值向量。你可以通过求变量向量和系数的点积（或线性组合）来预测目标值。

协同过滤

基于许多用户（协同）的集体偏好，自动提供推荐（过滤掉大量可能的项目）的方法。协同过滤技术通常是使用机器学习算法开发的模型，用于作出影响用户行为的预测。

collector

一种特殊的 Flume 代理类型，用于监听来自多个上游代理的数据，聚合其输出，然后将输出收集到日志文件、HDFS 或 HBase。

列族

HBase 中的一组相关列，共享相同的前缀，并且逻辑上和物理上都存储在一起。

面向列 / 列式数据库

内部按列（而不是按行）存储数据的数据库系统，尤其适合于在选定列执行聚合的 OLAP 用例。

满足交换律的

在数学中，无论运算以什么顺序执行，满足交换律的运算都返回相同的结果。满足交换律的运算在分布式计算中很重要，因为它允许数据以任何顺序进入，并返回相同的结果。

可比较的

具体是指可以使用不等式（例如大于）来比较两个对象。Java 和 Python 都提供了数据模型，通过定义必须返回不等式结果的方法，允许对象比较。

复杂键

不是简单或原始类型（例如整数或字符串）的键（例如键 / 值对中的键）。大多数复杂键是复合键的形式；其他类型的复杂键可以是嵌套数据类型（例如字典），或可以表示任意数据的字节数组。另请参见“复合键”。

复合键

由两个或多个简单键组成的键（例如键 / 值对中的键），通常存储在元组中。复合键对于 map 阶段和 reduce 阶段之间数据组织，以及作业之间的数据组织很重要。

计算

在本书中，指使用计算机处理器对某些数据执行计算。这里的计算与“存储”不同，它作用于数据输入并产生数据输出。

无冲突的复制数据类型（conflict-free replicated data type，CRDT）

一种特殊的数据结构，可以通过满足结合律和满足交换律的运算并发地变化。CRDT 在分布式系统中提供最终的一致性和单调性（不能回滚）。另请参见“累加器”和“计数器”。

一致性

分布式计算的属性，其中单个任务的失败不会影响最终结果；也表示分布式系统中的所有节点都会看到相同的数据视图。

列联表

具有两个维度的表格，允许检查分类变量之间的关系。表维度的交点（每个单元格）包含每个维度的分类值的共现频率。

计数器

在 MapReduce 中，计数器是一个共享变量，只能以固定的量递增。因为求和是满足结合律的，递增的顺序并不重要，所以可以在分布式环境中放心使用计数器。另请参见“累加器”。

守护进程

一种在后台运行的计算机软件，不需要用户输入。通常作为一种服务，监听网络上传入的信息并进行适当响应（服务器）。

数据分析师

主要关注数据产品开发的描述和推测的数据科学家，工作内容通常与建模、特征工程和探索相关。另请参见“数据建模师”。

数据应用程序

一种软件应用程序，用于处理大量的领域特定数据。例如，Microsoft Excel 是一种数据应用程序，用于处理电子表格或财务方面的数据。另请参见“数据产品”。

数据工程师

主要关注数据技术的数据科学家，工作通常与软件开发、数据库工具和计算基础设施有关。

数据流

在数据流中，一个单位的数据或事件（例如单个日志语句）从源流经一系列跃点到达下一个目的地。

数据湖泊

一个存储系统，用于以原始（被采集的）格式存储大量**原始**数据，一般采用无格式或半结构化格式。通常在数据湖泊中应用提取、转换和加载（ETL）操作来提取本地数据集市，将其用于下游计算。

数据本地计算

一种分布式计算概念，旨在减少所需的网络流量。节点对其本地存储的数据进行计算，而不试图从集群中的其他位置获取数据。

数据挖掘

分析不同来源的数据以生成新信息或获取更深入洞见的过程。

数据建模师

数据科学家的分支，根据统计和机器学习模型对数据进行探索和解释。

数据并行

一种在多个处理器之间进行计算的方法，其中数据分布在不同的节点上，各个节点同时对数据应用相同或相似的计算。

数据产品

自适应的、广泛适用的经济引擎，从数据中获取价值，通过影响人类行为或通过对新数据进行推论或预测，产生更多数据。

数据科学

创建和开发数据产品所涉及的工作流和过程。

数据科学流水线

描述数据科学分析过程的教学模式。流水线规定了一个线性过程，数据在其间被采集、整理、计算、建模并最终得以可视化。

数据科学家

他们是拥有强大统计学背景的程序员，是具备高超程序设计能力的分析师，是非常了解数据如何影响可视化的设计师，或是在构建数据产品方面富有创新思想的领域专家。在任何情况下，数据科学家都是全能的通才，能够轻松学习新的方法来处理数据。

数据仓库

大型数据存储，通常为关系型，包含一个组织多个维度或多个方面的数据。数据仓库通常以“星型模式”组织，以便在事务成本和在线异步处理之间取得平衡。另请参见“企业数据仓库”（enterprise data warehouse，EDW）。

数据库

简单来说，就是以电子格式存储的数据的集合。然而，它通常是“数据库管理系统”的缩写。数据库管理系统是一个软件应用程序，负责存储在磁盘上的数据的组织、管理和访问。

DataFrame

一种数据结构，指的是以行（案例或实例）和列（特征或度量）结构化的表格数据。DataFrame 从 R 编程语言就开始流行，并在 Python（通过 Pandas 库）和 SparkSQL（现在的 Spark DataFrame）中实现。

DataNode

指 HDFS 中，运行在集群中每个存储节点上的服务，提供数据复制。DataNode 与 NameNode 连接，以提供和分布式存储状态有关的信息，并响应客户端对文件系统操作的请求。

决策空间

指机器学习中，由实例特征给定的多个维度定义的空间中的一个区域，决策是针对实例特征的。决策空间越大，模型的通用性越强。另请参见“特征空间”。

声明式语言

指编程中的一种非命令式语言，让程序员在不明确列出从输入到输出应采取什么步骤的情况下描述所需结果。SQL 是一种声明式语言，而 Python 不是。

去规范化

通过分离关注点，规范化的数据存储在多张表中。去规范化指将这些数据描述为单张表的过程，通常通过使用 `JOIN` 函数。去规范化的数据以冗余为代价，形成集中的单个完整的记录。

反序列化

将数据（通常是软件中的对象）的字符串或字节表示加载或转换回可由程序使用的运行表示的过程。

分布式缓存

一种 MapReduce 工具，类似于 Spark 的广播变量。在执行任务之前，用于计算的所有节点所需的文件（例如停用词列表、查找表等）从 HDFS 被复制到每个 worker 节点。

分布式计算

一种软件或计算系统，其处理组件位于多台计算机上，互相之间通过网络通信，通过消息传递协调计算。分布式计算允许多台计算机并行工作，提供了性能优势。但是由于协调需求，它通常要求算法的结构针对分发专门设计。

分布式存储

数据存储在安装在多台主机上的多个磁盘上。为了访问数据，需要网络流量来定位和获取所请求的数据。分布式存储确保了数据本地计算会发生，因为数据已经位于将发生处理的地方了。此外，大多数分布式存储系统还复制数据，让多个主机存储数据的冗余副本，防止数据丢失。

领域专家

数据团队成员，对建模领域有深入了解。特征工程过程需要领域专家在预测性和系统方面的直觉来指导模型运行。领域专家通常也是敏捷开发的客户，为工程和分析过程提供指导。

企业数据仓库（EDW）

用于支持企业级数据报告和分析的中央数据存储库，被视为大多数商业智能环境的核心组件。

可执行文件

可以在计算机上执行的程序。Hadoop Streaming 可将可通过 `$PATH` 变量定位的可执行程序作为 mapper 或 reducer。例如，Python 可执行文件（拥有执行权限和指定解释器的 shebang 的 Python 脚本）可用于 MapReduce 编程。

执行计划

图或树，用于描述可执行过程或函数的顺序和数据流。Spark 应用程序通过一系列转换和一个最终被应用的动作来定义执行计划。SQL 和 HiveQL 是声明式语言，必须由底层系统转换为执行计划。

executor

在 Spark 中，executor 是运行在每个 worker 节点上的一个进程，代表集群管理器（YARN 上的 Spark ApplicationMaster）和驱动程序中的 SparkContext 来管理任务和数据服务。

容错性

如果一个组件出现故障，不应该导致整个系统出现故障。系统应优雅地降级到较低的性能状态。如果故障组件恢复，应该能重新加入系统。

特征空间

指机器学习中，由实例的属性（也被称为特征）定义的空间。特征空间还包括向更高维度的映射——对特征应用函数从而创建新值。例如，给定有 6 个因变量的一般线性模型，在特征空间中有 6 个维度来适应超平面。但对于二次多项式回归，由于平方映射将应用于原先的 6 个因变量，所以特征空间将有 12 个维度。另请参见“决策空间”。

过滤

在函数式编程中，过滤器是一个函数。它接受另一个函数和一个集合，并返回一个新的集合，新集合仅包含映射到过滤函数返回 `True` 的元素。换句话说，过滤函数是一个测试，确定一个元素是否属于一个较小的新集合。

第一范式

规范化数据库中的关系（表）的属性，使得每列只包含原子值，而每行的该列只包含一个值。例如，在这种范式下，属性不能是列表。

拟合模型

将超参数化模型形式拟合到数据的结果，通常通过优化函数，使模型参数能够基于新数据进行预测。拟合模型是机器学习训练的产物。

函数式编程

一种编程风格，计算在其中被视为函数的评估，用于避免状态改变或数据突变。因此，函数式编程是分布式计算的理想选择，因为需要固定状态和函数处理来确保协调一致。

可通用的

在机器学习中，如果一个模型可以根据未知的输入数据作出很好的预测，那么就称它为可通用的。如果一个模型拟合不足，那么就不能将该模型推广到更大的决策空间；如果一个模型过度拟合并仅是记住了数据，那么即使在本地决策空间中，它对未知数据的预测也是不正确的。

产生式模型

使用联合概率分布，而不是条件概率分布（判别式模型）来确定数据如何生成的模型。针对分类器，生成式模型对什么分类最可能产生信号作出了解答。

全局解释器锁（global interpreter lock，GIL）

解释型语言中的一种机制，用于同步线程的执行，确保任何时候只有一个线程在执行，以保护非线程安全的内存。GIL 是 Python 结构上的一部分，让 Python 先天不支持并发性；若想在 Python 中实现并行性，必须使用多个进程，且每个进程都有自己的 GIL。

Google 的 BigTable 架构

BigTable 是一种分布式存储系统，用于管理可以扩展到非常大的结构化数据。Chang 等人在 Google 2006 年的论文“Bigtable：A Distributed Storage System for Structured Data”中对此进行了详细讨论。

图形分析

一种分析，用于评估结构为以边相连的顶点的数据。顶点和边都可以包含数据，并且可以通过遍历来计算这种形式的数据集（而不是矩阵形式的数据集）。遍历本质上是可并行化的，因此，图形算法可以立即应用于大数据环境中。Spark GraphX 这样的库提供了图形分析工具。

Hadoop Pipes

一个内部 MapReduce 系统，允许 C++ 代码访问 HDFS 并执行 mapper 和 reducer。Pipes 与 Streaming 类似，都将 Pipes 代码分割为独立的应用程序特定的库。但与 Streaming 不同的是，Pipes 支持类型化的字节序列化和更全面的 API。

Hadoop Streaming

MapReduce 应用程序的实用程序，允许将任何可执行文件用作 mapper 和 reducer。Hadoop Streaming 本身就是一个 MapReduce 应用程序，它通过标准输入将数据传输到可执行文件，然后通过标准输出和标准错误从可执行文件中收集信息。Hadoop Streaming 让 Python 和 R 开发人员能编写 MapReduce 代码。

可散列的

在 Python 中，如果一个对象拥有一个在其生命周期内永远不会改变的散列值，那么该对象就是可散列的。因此，可散列的对象是不可变对象，或者是通过其内存地址进行散列的类的实例。所有可以用作字典中的键的内容都是可散列的（例如非列表或其他字典）。

HDFS

Hadoop 分布式文件系统，Hadoop 的两个主要组件之一。HDFS 通过在集群上实现三种类型的服务来提供分布式存储，分别是 NameNode、Secondary NameNode 和 DataNode。

高基数

指列或数据属性的值非常罕见或各不相同（每条记录都有一个值）。高基数的列难以分析或聚合，自动数据类型检测也通常不起作用。

Hive

一种为 HDFS 中存储的数据提供类 SQL 接口的系统。Hive 让数据科学家能将 Hadoop 视为分布式数据仓库，并能以结构化方式并行执行 OLAP 操作。

Hive CLI

Hive 命令行接口，与 Hue 打包在一起。它提供了一个交互式 shell，用于使用 Hive 并执行 HiveQL 语句。

Hive metastore

Hive 使用的数据库，用于存储 HDFS 上有关 Hive 表和分区的元信息。

HiveQL

Hive 查询语言，属于 ANSI SQL 子集的 Hive 数据定义语言（Data Definition Language，DDL）。

假设驱动开发

敏捷数据产品开发方法，用假设替代需求，并尝试使迭代敏捷开发过程与实验、观察和重新定义的迭代科学方法模型相匹配。

恒等函数

一个总是返回与参数相等的值的函数。在数学中，这表示为 $f(x) = x$。

不可变的

不随时间改变或无法改变。在 Python 中，不可变对象在运行时无法修改，例如元组、字符串、整数或布尔值。不可变对象提供了许多好属性，例如安全性（对象被传递给函数时不会发生意外突变）、紧凑性（所需内存减少）和散列的可比性。

索引

从较长形式的数据导出汇总数据结构的计算，以便快速查找各个记录。索引作为一个预处理步骤，用来提高下游计算的速度。

采集

数据采集是指从外部源收集数据并在本地计算环境管理数据的手动或自动过程。在大数据环境中，采集通常意味着并行处理输入数据流，以便及时获取数据。

输入 / 输出

在编程中，输入是提供给进程或函数用以计算的数据，计算的结果就是输出。通常，此形式中的输入 / 输出（input/output，I/O）是指从磁盘收集数据，将其发送到处理器，最后将结果写回磁盘的过程。

交互式分析

一种技术，将计算机在大量数据上进行重复任务的计算能力，与能够从全局层面识别模式和一般性的人类认知理解力相结合。交互式分析可以引导自动模型生成，或通过可视化来调整模型的行为。

倒排索引

一种特殊索引，将单词、数字、用户或重要信息等内容映射到它们在数据库、文件、文档或文档集合中的位置。

迭代计算

一种重复计算，其中单个计算块被定义为迭代；每次迭代重复，使上一个迭代的输出成为下一个迭代的输入。迭代和递归是计算机算法的基本构件。

迭代数据处理

迭代计算的一种，指一种算法对相同数据进行多次处理，将每个迭代的结果传递到下一次迭代，但不改变数据。优化就是迭代数据处理的一个例子，一次数据处理用于计算误差，下一次迭代修改参数以缩小误差，之后算法继续迭代，直到误差低于某个小阈值。

Java 数据库连接（Java database connectivity，JDBC）

基于 Java 的接口，允许客户端使用兼容的适配器访问支持 JDBC 的数据库。Sqoop 使用 JDBC 连接器与第三方数据库集成。

作业

在分布式计算中，作业是指完整的计算，由多个并行运行的独立任务组成。

作业链

MapReduce 应用程序中使用的一种技术，通过将前一个作业的输出用作下一个作业的输入，将一个或多个 MapReduce 作业链接在一起，从而构建更复杂的算法。

作业客户端

客户端是作业的发起者，是最关心结果的一方。客户端可以在作业运行期间保持连接，也可以先让作业在集群上独立运行，过后再返回以查找结果。

作业配置

用于定义范围的作业参数，例如应该使用的 mapper、reducer 或 executor 的数量。

Jupyter notebook

以前的 iPython notebook。notebook 是结合了可执行代码和富文本的文档，旨在以一种演示文稿的格式展示分析及其结果。因此，它们被广泛用于分析，从而显示可重现的结果。

Kerberos

用于验证服务请求的安全方法，可用于 HDFS 和 YARN API 以及集群保护。

键 / 值

一种联结的数据项，其中键是与数据值相关联的唯一标识符。键 / 值对将关系（由键定义）分发到多个处理器，然后聚合（reduce）其结果。

键空间

系统中用于计算的键 / 值对中键的域。键空间定义了数据如何分区到 reducer，以及键值对如何分组和比较。

lambda 架构

一种系统设计，能应对不断采集到的、需要使用分布式计算框架（如 MapReduce 或 Spark Streaming）及时处理的大量数据。lambda 架构使用消息队列前沿来缓冲传入到处理速度可能较慢的应用程序中的数据。这些应用程序执行初步计算并将结果存储在 speed 表中，执行最终计算并将结果存储在 batch 表中。客户端查询近似的 speed 表及时获取结果，依靠 batch 表进行更准确的分析。

lambda 函数

在 Python 中，lambda 关键字用于定义未绑定到标识符的匿名函数。另请参见“匿名函数”和“闭包”。

延迟执行

一种将表达式评估延迟到必要时刻的策略，从而达到最小化计算和重复性的目的。在 Spark 中，应用于 RDD 的转换操作通过生成转换过程（lineage）图来延迟执行，仅在对 RDD 应用动作操作时才执行。

词汇多样性

自然语言语料库中的单词数量与词汇量的比例，例如一个单词在语料库中的平均使用次数。词汇多样性用于监测文本数据的异常变化。

转换过程（lineage）

在 Spark 中，每个 RDD 存储通过应用转换从其他数据集构建出它的机制。转换过程让 RDD 能在故障时本地重建，并为 Spark 中的容错性提供基本机制。

线性作业链

一种作业序列，前一个作业的输出用作后一个作业的输入。另请参见“作业链”。

log4j

一个开源 Java 项目，允许开发人员控制日志消息输出的粒度。在 Spark 或 MapReduce 中修改 log4j 设置可以最小化控制台输出的数量，使分析人员能更轻松地了解结果。

机器学习

发现数据模式，利用这些模式构建模型，对新数据进行预测或估计的技术。

map

一种函数式编程技术，其中函数被应用于集合中的每个单独元素，生成一个新集合作为每个 map 的输出。map 本质上是可并行的，因为将 map 函数应用于元素不依赖于任何其他 map。

master 节点

集群中的一个节点，实现了一个主守护进程（用于管理集群中的存储和计算的进程）。主进程包括 ResourceManager、NameNode 和 Secondary NameNode。

最大值

在描述性统计中，数据集中的最大值。

平均值

在描述性统计中，通过将数据集中数值之和除以值的数量，来描述数据的集中趋势的值。

中位数

在描述性统计中，有序数据列表中的中间值。

微框架

一个术语，指简约的应用程序框架。本书使用 Python 和 Hadoop Streaming 构建了一个 MapReduce 的微框架。

最小值

在描述性统计中，数据集中的最小值。

众数

在描述性统计中，数据集中最常出现的值。

模型族（model family）

在机器学习中，模型族从宏观角度描述了决定预测的相关变量之间的联系。例如，基于

系数向量与因变量向量的线性组合，线性模型描述了对连续目标值 Y 的预测。

模型形式（model form）

在模型拟合之前对模型轮廓的说明，特别定义了超参数，以及拟合模型的特征空间。例如，给定支持向量机模型族，模型形式可能是拥有 RBF 核函数、伽马值为 0.001 且松弛变量为 1 的 SVM。

munging

munging 最初来自麻省理工学院模型训练小组，指的是将数据混合在一起形成的统一或规一化整体，可能具有破坏性。

NameNode

HDFS 的 master 节点，负责集群 DataNode 的集中协调。NameNode 分配存储资源，并将大文件拆分成块，在集群中复制。NameNode 还将客户端直接连接到要访问数据的 DataNode。

节点

集群中的单个机器，能实现一些服务，特别是守护进程服务（如 NodeManager 和 DataNode）。

NodeManager

YARN 中在集群中的每个节点上运行的进程或代理。NodeManager 负责跟踪和监视各个 executor（容器）的 CPU 和内存的使用情况以及节点的运行状况，并将其报告回 ResourceManager。NodeManager 还通过调度 executor（容器）在本地进行工作，来代表 ApplicationMaster 执行框架作业。

NoSQL

指“不仅仅是 SQL”（not only SQL）或者“非关系型”（not relational）。NoSQL 最初是一个标签，在一场讨论数据库技术（如 Cassandra、HBase 和 MongoDB）的会议中使用。NoSQL 现在指不符合传统关系数据库管理系统定义的数据库，它们提供的领域特定数据模型（如图或列）通常是分布式的。

大数据操作系统

Hadoop 的两大支柱服务——HDFS 提供分布式数据存储、YARN 提供集群计算资源管理——为集群计算提供了平台，也使它成为了大数据的操作系统。

运行阶段

在机器学习中，构建阶段之后就是运行阶段。拟合模型在此阶段进行预测（回归分析作出连续值估计、分类器分配类别、聚类确定归属）。

上线

在数据产品中使用拟合模型。另请参见“运行阶段”。

pairs 和 stripes

在矩阵（例如单词共现矩阵）上执行分布式计算的两种方法。在 pairs 方法中，矩阵中的行 i 和列 j 的每个单元格单独映射，如 (i, j)/ 值；在 stripes 方法中，每行 i 作为完整的值被映射，通常作为列 j 的关联数组。

Pandas

一个开源库，提供易于使用的数据结构，如 Series 和 DataFrame，可以应用多个数据分析工具。

并行

两个计算同时运行。

可并行的

如果可以将一个算法分解成可以同时运行的离散任务，则该算法被称为可并行的。可并行的算法有一个属性：可并行运行的任务越多，算法的完成速度就越快。

并行化

算法向可并行形式转化的过程。

对等集群

与集中管理的集群相反，对等集群是完全去中心化的，没有控制源。执行对等协调的算法不能依赖于一个集中的控制中心。Hadoop 和 Spark 是集中管理的集群，而 Bitcoin 这样的应用程序是完全去中心化的，被称为对等分布式计算。

Pig

Pig 是一个大数据框架，由 Pig Latin（一种用于表达数据分析程序的高级语言）和一个编译器组成，后者可以将 Pig Latin 翻译成在 Hadoop 上执行的 MapReduce 作业序列。

POSIX

“可移植操作系统接口”（portable operating system interface）是 IEEE 计算机学会创建的一系列标准，旨在提高操作系统之间的兼容性。

预测模型

一种统计工具，通过推断技术来描述未来可能发生的行为。

过程化语言

与声明式语言相反，过程化语言定义了一个必须逐个执行的有序命令集。Python 可以使用过程化风格来编写，当然用函数式风格或面向对象风格也没问题。

进程

进程是正在执行的计算机程序的实例，包括完整的计算环境和资源。进程可以由并发运行的多个执行线程组成；但是一般来说，当在分布式环境中讨论进程时，是指一个必须通过网络与其他程序通信的独立程序。

产品印象

在线营销术语，指单一用户有机会查看特定的产品，通常是与超链接相关联的产品。通过数据采集技术，我们可以通过比较生成产品印象的网络日志及相关的点击率来监控产品印象是否成功。

投影

投影是一种操作，针对由一组属性定义的关系（表）。投影输出一个新关系，丢弃或排除原始关系中不在投影里的属性；换句话说，投影会移除表中的列。

PySpark

交互式 Python Spark shell，被实现为命令行 REPL（read, evaluate, print loop，读取、评估、打印循环），并由 `pyspark` 命令启动。

Python Spark 应用程序

由 Python 编程语言编写，调用 Python Spark API，通过 `spark-submit` 提交到 Spark 上运行的应用程序。

随机访问

指在一组可寻址元素内的任何给定内存地址访问特定数据项的能力，这与按照磁盘顺序读取数据元素的顺序访问相反。

推荐系统

一个信息系统，目标是预测用户对某些项目的评级或偏好。推荐系统通常使用协同过滤算法实现，它根据类似的用户偏好对项目的整个空间进行过滤。然后使用非负矩阵分解和回归模型等机器学习技术对评级进行预测。

可恢复性

分布式系统的属性，使数据不会在故障发生时丢失。

关系

关系是一组元组，元组的每个元素都是数据域（或数据类型）的成员。在数据库系统中，关系通常是指一个表，行的列都是带类型的。

关系数据库管理系统（relational database management system，RDBMS）

根据数据库、表、列、关系的关系型建模原则组织数据的数据库系统。RDBMS 中的查询操作通常使用 SQL 查询语言的某种变体。

水库抽样

一系列随机算法，用于从 n 个项目的列表中随机选择 k 个样本，其中 n 是非常大的数或未知数。

弹性分布式数据集（resilient distributed dataset，RDD）

Spark 中的基本抽象，代表可并行操作的、不可变的、分区的元素集合。

ResourceManager

YARN 中的一个主进程，通过按需给 ApplicationMaster 分配资源（可用的 NodeManager executor 实例），来调度集群上的计算工作。基于预配置策略，ResourceManager 尝试通过容量保证、公平性或服务级别协议来优化集群利用率（让尽可能多的节点肯能忙）。

岭回归

线性回归模型族中的一个正则化模型，通过使用系数的 L2 范数正则化误差最小化函数来惩罚模型复杂性（从而减小模型的偏差）。使用 L2 范数会使权重平滑，降低多重共线性导致的方差的影响。

行键

在 HBase 中，按照唯一的行键对行进行访问和排序。行键本身只是一个字节数组，但是良好的行键设计是在设计健壮的 HBase 数据库数据访问模式时最重要的考虑因素。

可扩展性

分布式系统的属性，使负载增加（更多的数据，更多的计算）将导致性能下降，而不是故障；资源增加，容量也成比例增加。

Secondary NameNode

Secondary NameNode 每隔一段时间复制一次主 NameNode 镜像的编辑日志，定期创建 HDFS 的检查点。它不是主 NameNode 的替代品或备份，而是在重新启动时能让 NameNode 更快恢复。

自适应

一些机器学习模型的属性，可以用新的信息增量更新。数据产品本身应该是自适应的，但若没有增量更新，就得要重新训练模型。

可分的

数据的一种属性，让类可以在特征空间中被超平面分割或分离，具有一些松弛。可分性意味着像支持向量机和随机森林这样的模型将是非常有效的。

序列化

在数据存储上下文中，序列化是一个过程，能将数据结构或对象状态转换为可以存储（例如，存储在文件或内存缓冲器中，或者通过网络连接链路传输），并且之后能够在同一台或别的计算机环境中重新构造回来的格式。

shebang

在脚本起始处的字符序列 #!，由字符数字符号和感叹号组成 。

单节点设置

在 Hadoop 中，单节点设置在单个机器上安装所有进程（比如 YARN、HDFS、JobHistory Server 等）。也被称为伪分布式设置。

数据槽

数据流中流入数据的接收器或目标。

数据源

可以是数据库、数据存储设备或者进程，发送馈入数据流的输出数据，用于进一步处理或传输到数据槽。

垃圾信息

未经请求和不需要的消息或电子邮件。

Spark Core

构成 Spark 根本的程序内部和抽象（包括 RDD API）的组件、服务和 API。

Spark Python API

Spark 公开的 Python 语言的应用程序编程接口，用于创建 Spark 应用程序。它提供了对 Python RDD 以及 Spark 中许多库工具和代码的访问。

稀疏的

表示数据中有相当高比例的值或单元格不包含实际数据或为“null”。

推断执行

一种能最小化延迟或失败作业的影响的技术。如果检测到慢任务，则立即在相同数据上分配新任务；获胜者是首先完成的任务。

分割

基于某些标准，将数据集划分为多个子集的过程。

分段

将数据传输到中间数据目标或检查点供后续处理的过程。

独立模式

在 Spark 中，此模式可用于在本地计算机的单个进程中运行 Spark。

流数据

不间断或无边界的数据流，稳定且持续地传输和处理。

stripes 和 pairs

请参见“pairs 和 stripes”。

主题专家

主题专家是数据科学家，是数据团队的关键成员。他们为数据问题和模型提供特定领域的知识。另请参见“领域专家”。

有监督的

与无监督机器学习相反，有监督机器学习将模型拟合到数据集，正确答案是提前知道的。分类和回归是有监督机器学习的两个例子。

任务

一个 YARN 作业中的工作单位。在 MapReduce 中，任务是指执行一次 map 操作或 reduce 操作。

任务并行

一种并行形式，在相同或不同的数据集上同时执行多个函数来提升性能。它与数据并行相反，后者是将相同的函数应用于数据集的不同元素。一般来说，map 是数据并行，reduce 是任务并行。

大数据的 3V

定义大数据的 3 个属性：容量（volume）、速度（velocity）和多样（variety）。另请参见“容量”“速度”和“多样”。

转换和动作

指两种主要 Spark 操作。转换将 RDD 作为输入，并产生重新格式化的 RDD 作为输出；动作在 RDD 上执行计算，产生一个值返回到 Spark Driver。

元组

一个有限的、不可变的有序元素的集合。

无监督的

与有监督机器学习相反，无监督机器学习基于模式通过实例之间的相似性或距离来拟合

模型。这些模型族是无监督的，因为没有“正确的”答案来检验拟合模型的结果或者使误差最小化。聚类是无监督学习的一个例子。

方差

在机器学习中，方差是指给定特定数据点的模型的预测可变性（例如低方差可能表示对于预测的误差量的置信度）。随着方差的减小，偏差将增大。另请参见“偏差”。

多样

数据的结构化格式（CSV、Excel、数据库等）和非结构化格式（图像、传感器数据、视频等）的范围不断扩大。

速度

处理数据的速度或速率。

词汇量

一个文本语料库中唯一令牌（或单词）的集合。

容量

待处理和存储的数据的量。

worker 节点

实现 worker 守护进程的节点，worker 守护进程通常都是 NodeManager 和 DataNode 服务。

工作流管理

构建可触发、可参数化、可调度、可自动化的可重复数据处理作业的过程。

数据整理

将数据从一种格式（通常为“原始”未处理的格式）转换或映射到另一种格式的过程，使得下游进程可以轻松消费数据以进行分析。

YARN

“Yet Another Resource Negotiator”的缩写，是包括 MapReduce 和 Spark 在内的分布式计算引擎的广义集群管理框架。它将处理提交给集群的作业的资源管理和作业调度。

关于作者

Benjamin Bengfort 是一位数据科学家。他虽然住在市里，却对政治（华盛顿的正事）毫无兴趣，反而偏爱技术。他目前正在马里兰大学攻读博士学位，在那里学习机器学习和分布式计算。他的实验室里确实有机器人（虽然这不是他喜欢的研究领域），但令他懊恼的是，他们老是想给机器人配上刀具或者餐具——大概是为了什么烹饪大奖吧。与其看一个机器人切番茄，Benjamin 更喜欢自己在厨房里研究，他热衷于将法国和圭亚那美食融合，也喜欢各种类型的烧烤。Benjamin 是一名专业编程人员，但他把数据科学家当作自己的事业。他的作品涉及大量主题，从自然语言处理，到使用 Python 的数据科学，再到 Hadoop 和 Spark 分析，应有尽有。

Jenny Kim 是一位经验丰富的大数据工程师。她开发商业软件，也在学术界工作，在海量数据、机器学习以及生产和研究环境的 Hadoop 实施方面有深入研究。Jenny（和 Benjamin Bengfort 一起）建立了一个大型推荐系统，使用网络爬虫收集服装产品的有关信息，并根据交易提供推荐。目前，她正与 Cloudera 的 Hue 团队一道，试图构建能用于 Hadoop 大数据分析的直观界面。

关于封面

《Hadoop 数据分析》的封面动物是牛背鹭（Bubulcus ibis），一种遍布世界的白鹭。起初，牛背鹭只生活在亚洲、非洲、欧洲的部分地区，但在 20 世纪，它已经“入侵了”地球其他大部分区域——经历了一次鸟类最快速、最广泛的自然扩张之后。它主要在热带、亚热带和温带栖息，经常跟随牛或其他大型哺乳动物生活，以这些大型动物驱赶的昆虫或小型脊椎动物为食，因此得名。

牛背鹭是一种白色鸟类。背部、前颈和头部在繁殖期呈橙黄色；与配偶交配之前，喙、腿和虹膜会暂时变成亮红色。非繁殖期的成年牛背鹭几乎通体白羽，黄色喙，灰黄色腿。它身材矮壮，翼展 35 英寸 ~38 英寸（约为 89 厘米 ~97 厘米），体高 18 英寸 ~22 英寸（约为 46 厘米 ~56 厘米），重约 10 盎司 ~18 盎司（约为 283 克 ~510 克）；通常在水边的树木和灌木丛的树枝上筑巢。

因为与牛存在共生关系，牛背鹭深受牧场主的欢迎，被认为是防治牛身上寄生虫的生物手段。但另一方面，当大群牛背鹭在机场的草地边觅食时，可能会造成飞机的安全隐患。此外，它们还可能传播动物疾病，如心水病、传染性法氏囊病甚至新城疫。

O’Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

封面图片来自 *Lydekker’s Royal Natural History*。

微信连接

回复“大数据”查看相关书单

微博连接

关注@图灵教育 每日分享IT好书

QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

O'REILLY®

Hadoop数据分析

通过提供分布式数据存储和并行计算框架，Hadoop已经从集群计算的抽象演变成了大数据操作系统。本书从数据科学的视角，介绍Hadoop集群计算和分析，重点关注可构建的具体分析、数据仓储技术和高阶数据流。

书中主要内容如下：

- Hadoop和集群计算背后的核心概念
- 使用设计模式和并行分析算法创建分布式数据分析作业
- 在分布式环境下使用Apache Hive和HBase进行数据管理、数据挖掘和数据仓储
- 使用Sqoop和Apache Flume从关系数据库采集数据
- 使用Apache Pig和Spark DataFrame编写复杂的Hadoop和Spark应用程序
- 通过Spark MLlib运用分类、聚类和协同过滤等机器学习技术

"我此前还未见过比本书更好的Hadoop框架讲解。"

——Marck Vaisman
博思艾伦咨询公司数据科学家、
乔治·华盛顿大学兼职教授、
数据社区DC联合创始人

"每个概念都得以清晰明了的解读，在容易忽略细节的部分又都有补充资源，供读者深入学习，这对于专业人员和初学者都非常友好。本书中的讲解总是与示例相辅相成，让读者在学习之后又能投入实战，深入了解系统功能——我认为这才是熟悉新领域的关键所在。"

——Amazon读者

Benjamin Bengfort，数据科学家，目前正在马里兰大学攻读博士学位，方向为机器学习和分布式计算；熟悉自然语言处理、Python数据科学、Hadoop和Spark分析等。

Jenny Kim，经验丰富的大数据工程师，不仅进行商业软件的开发，在学术界也有所建树，在海量数据、机器学习以及生产和研究环境的Hadoop实施方面有深入研究。目前就职于Cloudera的Hue团队。

DATA

封面设计：Randy Comer 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / 大数据

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-47964-8

9 787115 479648 >

ISBN 978-7-115-47964-8
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks